2

US Army Corps of Engineers
Construction Engineering Research Laboratory

USACERL Special Report P-91/46
September 1991
Functional Analysis Model Research Platform

Application of Object-Oriented Programming to Combat Modeling and Simulation

by
Charles Herring
R. Alan Whitehurst
Jeffery Wallace
David Adams

The Department of Defense (DOD) has defined modeling and simulation as an area of technology critical to long-term U.S. national security. Research in this area by the U.S. Army Construction Engineering Research Laboratory (USACERL) has included the development of the Force Structure Tradeoff Analysis Model (FSTAM) combat engineering modeling and simulation program for the U.S. Army Engineer School (USAES). Recent advances in object-oriented programming and hypertext systems have helped stimulate interest in upgrading FSTAM.

As part of a separate-but-related initiative, Analysis Command of the U.S. Army Training and Doctrine Command (TRADOC) sponsored the development of ModSim, an object-oriented programming language for simulation applications. This language was recognized as a highly valuable tool for application in the planned upgrade of FSTAM.

This report describes USACERL's application of object-oriented programming techniques to the development of combat simulations—specifically, combat engineer functional representations. Included is discussion of current research prototypes of FSTAM 2.0 and an enhanced version of ModSim.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE September 1991	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE

Application of Object-Oriented Programming to Combat Modeling and Simulation

5. FUNDING NUMBERS

PE 4A162784
PR AT41
TA SE
WU YC1

6. AUTHOR(S)

Charles Herring, R. Alan Whitehurst, Jeffery Wallace, and David Adams

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

U.S. Army Construction Engineering Research Laboratory (USACERL)
PO Box 9005
Champaign, IL 61826-9005

8. PERFORMING ORGANIZATION REPORT NUMBER

SR P-91/46

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

HQUSACE
ATTN: DAEN-ZCM
20 Massachusetts Avenue, NW.
Washington, DC 20314-1000

US Army TRADOC
ATTN: ATRC-FPE
Ft. Leavenworth, KS 66027

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

Copies are available from the National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

The Department of Defense (DOD) has defined modeling and simulation as an area of technology critical to long-term U.S. national security. Research in this area by the U.S. Army Construction Engineering Research Laboratory (USACERL) has included the development of the Force Structure Tradeoff Analysis Model (FSTAM) combat engineering modeling and simulation program for the U.S. Army Engineer School (USAES). Recent advances in object-oriented programming and hypertext systems have helped stimulate interest in upgrading FSTAM.

As part of a separate-but-related initiative, Analysis Command of the U.S. Army Training and Doctrine Command (TRADOC) sponsored the development of ModSim, an object-oriented programming language for simulation applications. This language was recognized as a highly valuable tool for application in the planned upgrade of FSTAM.

This report describes USACERL's application of object-oriented programming techniques to the development of combat simulations—specifically, combat engineer functional representations. Included is discussion of current research prototypes of FSTAM 2.0 and an enhanced version of ModSim.

14. SUBJECT TERMS

object-oriented programming
models
warfare
simulation

15. NUMBER OF PAGES

56

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR

FOREWORD

This study was conducted for the Office of the Chief of Engineers (OCE) under Project 4A162784AT41, "Military Facilities Engineering Technology"; Task Area SE; Work Unit YC1, "Functional Analysis Model Research Platform." The OCE technical monitor was Mr. Mike Shama, DAEN-ZCM. This report also describes work done for the Model Improvement and Study Management Agency (MISMA), Analysis Command, U.S. Army Training and Doctrine Command (TRADOC), under the reimbursable Work Unit RD-P69-HR0, "FAFS-AMIP SimTech ModSim Data Management." The TRADOC technical monitor was Mr. Harry Jones, ATEN-FPE.

This research was performed by the Facility Systems Division (FS) of the U.S. Army Construction Engineering Research Laboratory (USACERL). Dr. Michael J. O'Connor is Chief of FS. The USACERL technical editor was Gordon L. Cohen, Information Management Office.

COL Everett R. Thomas is Commander and Director of USACERL, and Dr. L.R. Shaffer is Technical Director.

CONTENTS

FIGURES

APPLICATION OF OBJECT-OREINTED PROGRAMMING TO COMBAT MODELING AND SIMULATION

1 INTRODUCTION

Background

Congress requires the Department of Defense (DOD) to prepare annually a Critical Technologies Plan[1] that identifies the most essential technologies required to ensure long-term U.S. national security. On the list of the most critical technologies, categorized as an "enabling" technology, is modeling and simulation. Research in this area by the U.S. Army Construction Engineering Research Laboratory (USACERL) has included the development of combat engineering modeling and simulation technologies in support of the U.S. Army Engineer School (USAES).

The first phase of USACERL's combat modeling and simulation research investigated the feasibility of developing a microcomputer-based analysis combat simulation for use by USAES. The problems addressed during this initial phase were (1) to determine if a microcomputer could support a useful combat simulation and (2) to provide USAES with a base-capability model related to combat engineer functional activities. In July 1987, Version 1.0 of the Force Structure Tradeoff Analysis Model (FSTAM)[2] was delivered to USAES. This combat simulation runs on a microcomputer and is written in FORTRAN.

With accomplishment of the initial goals, USAES requested research into more advanced modeling and simulation technologies that would take advantage of recent progress in simulation and software development. Two factors having a potentially major impact on simulation technology are (1) a paradigm shift in software engineering from structured to object-oriented programming and (2) the general availability of hypertext systems.

As part of a separate-but-related initiative, Analysis Command of the U.S. Army Training and Doctrine Command (TRADOC) sponsored the development of an object-oriented programming language—ModSim—for simulation applications. This language was recognized as a highly valuable tool to employ in the planned upgrade of FSTAM.

Objective

The objective of the research described in this report is to explore the application of object-oriented programming techniques to the development of combat simulations—specifically, combat engineer functional representations. Included in this research is the application of hypertext as a medium for both the development and use of the simulation. The ultimate goal of this research is to develop an experimental object-oriented simulation platform for the testing and modeling of programming concepts as they apply to combat engineer representations.

[1] DOD *Critical Technologies Plan* (DOD Office of the Deputy Under Secretary for Acquisition, 1990).

[2] C. Subick and M.A. Fiddes, *Force Structure Tradeoff Analysis Model (FSTAM) User's Manual*, Automatic Data Processing (ADP) Report P-88/02 (U.S. Army Construction Engineering Research Laboratory [USACERL], August 1988).

Approach

Object-oriented programming was initially developed to provide high-fidelity mapping of real-world objects into software constructs for simulation, but its use has become the *de facto* methodology in software development. The Army developed the ModSim programming language in recognition of the limitations of current simulation languages and the trend toward object-oriented design and programming. The research team is using ModSim to develop a new version (2.0) of the FSTAM combat simulation. As a first step in this process, a hypertext system was used to capture the structure of the existing FORTRAN-based FSTAM 1.0. This effort provided insight into how hypertext representation of a simulation could provide an environment for development, maintenance, and user documentation.

Work also began on a major technical problem in large simulations: managing the vast amounts of data required for scenario development and generated for analysis. Traditional languages used for simulation lack the consistency of internal data representation required for a more intimate interface to a data manager. Object-oriented languages provide such an interface and the trend has been to include data management facilities directly in the language. This provides high efficiency of data retrieval and storage. The researchers have developed a research prototype version of ModSim that includes integrated data management facilities.

Mode of Technology Transfer

The enhanced combat simulation program FSTAM 2.0 will be delivered to USAES for maintenance and support. An upgrade of ModSim will be delivered to the Analysis Command Model Improvement and Study Management Agency (MISMA) for maintenance and support.

2 THE MODSIM PROGRAMMING LANGUAGE

Object-oriented design and programming are of particular interest to the simulation research community. Object-oriented design is the process of identifying the real-world objects comprising the system being modeled. Object-oriented programming languages provide the ability to faithfully represent these real-world objects in a computer simulation. The first object-oriented language was SIMULA 67,[3] which evolved from the simulation language SIMULA I. Thus, object-oriented languages and simulation have been related from the beginning.

In recognition of the trend toward object-oriented programming and its appropriateness for modeling and simulation, the U.S. Army Model Improvement and Studies Management Agency (MISMA) has sponsored the development of an object-oriented language for simulation. This new language is called ModSim (for modular simulation language). The following sections discuss the requirements for object-oriented programming languages, ModSim as an object-oriented language, and its facilities for supporting simulation.

Object-Oriented Language

Object-oriented design and programming represent a major paradigm shift in software engineering during the 1980s. With the advent of any new technology comes the inevitable period during which concepts are refined and terminology is clarified. Object-oriented programming is now in such a state of fluctuation. While there are many different approaches to building object-oriented languages, certain essential features are required for a language to be considered object-oriented.

Two concepts central to object-oriented design and programming are *class* and *object*. In general, a class is a set of things that have the same attributes and behaviors, e.g., cars. In general, an object is a specific example (instance) of a class, e.g., Bill's car. These two concepts have different connotations for design and programming. In design, the general meanings of class and object are used. In programming languages, class and object take on specific meanings. A class is a software unit (module) that encapsulates the attributes and behaviors of a real-world class identified in design. An object is an instance or realization of an item of a particular class.

As object-oriented languages have evolved, four major characteristics have emerged: *information hiding, data abstraction, inheritance,* and *dynamic binding*.

Information hiding was a significant advance in software concepts during the 1970s and is generally credited to Parnas.[4] According to his concept of information hiding, each software unit encapsulates its data and procedures and permits access to its internals only through a well specified interface. In object-oriented programming, objects communicate with other objects by sending messages requesting them to perform their behaviors. Objects can query other objects to find the value of an internal state variable, but they cannot directly change its value. Information hiding is implemented in programming languages by giving objects the ability to store both data structures and operations in a single module, which provides access through a specification interface and is protected by scope rules from direct manipulation by other objects.

[3] O.J. Dahl, B. Myrhaug, and K. Nygaard, *SIMULA 67 Common Base Language* (Norwegian Computing Center, Oslo, 1984).
[4] D. Parnas, "On the Criteria to be Used in Decomposing Modules," *Communications of the ACM*, Vol 15, No. 2, pp 1053-58.

Data abstraction is implemented in a language as the ability to create new types, of which variables are declared. In object-oriented languages, this is the ability to declare an object to be of a certain class. Class corresponds to an abstract data type while object corresponds to a variable of that abstract type. Data abstraction provides the ability to focus on what is relevant in modeling a problem.

Inheritance describes the mechanism by which classes are defined in terms of other classes. Through inheritance, classes can be created that are specializations or extensions of existing classes. The newly created class inherits the data structures and behaviors of its ancestor class. These derived classes are referred to as *subclasses*. The process of inheritance can be extended to many generations of subclasses. A familiar example is the class of mammals with its subclasses (orders, families, etc.) of other mammals representing specializations.

In programming, *binding* refers to the time at which values are associated with variables. The declaration of a constant, pi=3.1416, is called *early binding*—it is done when the code is written. *Dynamic binding* occurs when values are determined at run-time. Dynamic binding delays the association of data structures and behaviors until run-time. In object-oriented languages, the type of each operand and operation is determined at run-time.

A feature closely related to dynamic binding is *polymorphism*, which places the responsibility for correct action on the object. For example, in the mammal analogy given above, all mammals can move, but they do so in different manners. Some move on two legs, some on four, and some swim. Thus, the message "move," sent to different objects, will elicit different, but appropriate, behaviors.

A fifth requirement placed on object-oriented languages by some authors is *multiple inheritance*.[5] This feature permits classes to be created through inheritance from multiple parent classes.

ModSim as an Object-Oriented Language

ModSim is a general-purpose, block-structured, object-oriented programming language. The modular structure of the language and its syntax is based on Modula-2,[6] which is a direct descendant of Pascal. ModSim is not an exact superset of Modula-2, however. For a detailed comparison of the two languages, see Belanger and Rice.[7] For a complete treatment of the language, see Mullarney.[8] The purpose here is to discuss the object-oriented capabilities of ModSim as they relate to the criteria given above. (Note: the completely capitalized words in the following discussion are ModSim-reserved words.)

A ModSim program consists of a MAIN module and any number of library modules. Library modules consist of two parts: a DEFINITION and an IMPLEMENTATION. These modules are stored in separate files and are compiled separately.

A new class is defined through the use of the TYPE statement as follows:

```
TYPE
  CombatUnit = OBJECT
      Personnel : INTEGER;
```

[5] B. Meyer, *Object-Oriented Software Construction* (Prentice Hall, 1988).
[6] N. Wirth, *Programming in Modula-2* (Springer-Verlag, 1982).
[7] R. Belanger and S. Rice, *ModSim User's Manual* (CACI Products Company, 1988).
[8] A. Mullarney, J. West, R. Belanger, and S. Rice, *ModSim Tutorial* (CACI Products Company, 1988).

```
        Location,
        Destination : Coordinate;
        Speed : INTEGER;
    END OBJECT;
```

The new class identified as CombatUnit is declared similar to the Pascal record structure. One of the attributes, Coordinate, is a user-defined type declared elsewhere within the scope of this class.

ModSim has two *types* of *methods*: ASK and TELL. CombatUnit is expanded to include *methods* as follows:

```
TYPE
    CombatUnit = OBJECT
        Personnel : INTEGER;
        Location,
        Destination : Coordinate;
        Speed : INTEGER;
        ASK METHOD Status : INTEGER;
    END OBJECT;
```

CombatUnit now has one method—Status—that returns an INTEGER value. The use of TELL methods will be covered in the next section. In ModSim, all class declarations appear in the DEFINITION module, and many classes can be defined within one module.

The IMPLEMENTATION module contains the code for the methods. The details of the CombatUnit class would appear as follows:

```
OBJECT CombatUnit;
    ASK METHOD Status : INTEGER;
    BEGIN
        implementation code
    END METHOD;
END OBJECT;
```

This overview of ModSim's object implementation is necessary to examine information hiding. Information hiding is implemented at several levels. The separation of interface specification (DEFINITION) and code implementation (IMPLEMENTATION) permits information hiding and facilitates software reusability through separate compilation. An advantage of separate compilation is that source code for the DEFINITION can be supplied while the IMPLEMENTATION is supplied only in object form. This physically prevents the temptation to alter the original authors' intent and, thereby, the integrity of the design. The integration of objects into this scheme is quite natural, with the class declaration being contained in the DEFINITION module and the code for class behaviors in the IMPLEMENTATION module.

A second level of information hiding is in the scope rules of object visibility. ModSim's scope rules restrict access of attributes and behaviors to those specified in the DEFINITION modules. This provides the data and procedure encapsulation required of objects. An additional information hiding capability is the PRIVATE statement. It is used to restrict access of an object's data and behaviors to methods within the object itself.

Abstract data types are implemented by the TYPE and VAR statements, just as in Modula-2 and Pascal. ModSim introduces the class declaration in the DEFINITION module, e.g., TYPE CombatUnit = OBJECT. Objects are realized as variables of a class (TYPE) by declaration in the variable (VAR) section of a module.

In the example given above, the class CombatUnit was declared to be of TYPE OBJECT. It is through the TYPE construct that ModSim implements inheritance. Another example of inheritance in ModSim is shown in the code fragment:

```
TYPE
    ArmorUnit = OBJECT( CombatUnit );
     Tanks : INTEGER;
END OBJECT;
```

Notice the use of CombatUnit in the TYPE declaration as a parameter for OBJECT. This specifies that the new class ArmorUnit will inherit all of the attributes and methods of the class CombatUnit and is further refined by the addition of the Tanks attribute. In ModSim, complex derived classes can be constructed through use of inheritance of multiple path class hierarchies.

An object of the class ArmorUnit is realized by declaration as a variable of type ArmorUnit. The ModSim standard procedure NEWOBJ is called to allocate storage for object variables.

```
VAR
    ArmorPlatoon : ArmorUnit;
.
.
.
NEWOBJ( ArmorPlatoon );
```

The object ArmorPlatoon is now available to the program. Its attributes can be assigned values and it can interact with other objects.

Other features of ModSim related to objects include the ability to: (1) OVERRIDE inherited methods, (2) restrict the scope of attributes and methods though use of the PRIVATE statement, and (3) form groups of related objects (collection classes). ModSim uses dynamic binding and provides for polymorphic behavior, as do all true object-oriented languages. ModSim also supports multiple inheritance—the definition of classes in terms of more than one base type. The declaration of a class from two base types is shown below:

```
TYPE
    ArmorBattalion = OBJECT ( ArmorUnit, BattalionHeadQuarters );
...
END OBJECT;
```

Classes defined in this way inherit all attributes and methods of their parent classes. This opens the possibility for ambiguity in attribute and method names. ModSim resolves these conflicts by requiring direct reference to multiply defined attributes, and through use of the OVERRIDE statement for ambiguous methods.

ModSim as a Simulation Language

There are two types of discrete-event simulation: event-oriented and process-oriented.[9] *Event-oriented* simulations are based on the sequencing of a dynamic event list. Each event has an associated routine that determines when other events are placed on the event list. During the execution of an event routine simulation time does not advance. The event list manager advances time when the next scheduled event occurs. In *process-oriented* simulation, a process is a sequence of logically related activities ordered in time. The routine implementing the process contains all of its related activities. Each process maintains its own activity list. The system maintains a master activity list containing the next activity from each process' activity list. Clearly, the process-oriented strategy is more appropriate for object-oriented simulation. ModSim is based on the process model.

Process-oriented simulation in ModSim is inherently supported through addition of simulation primitives that permit time-elapsing methods. Classes can have multiple, concurrent activities. There are provisions for activities to operate synchronously or asynchronously and to interrupt activities within the same object or in other objects. This section describes the primary ModSim constructs for object-oriented process-based simulation: simulated time, the TELL METHOD, the WAIT statement, and the class TriggerObj.

Simulated time is a REAL (floating-point) value maintained by the ModSim run-time manager. It is dimensionless and can be used to represent any time resolution desired in a simulation. It is accessed by the function SimTime().

The code fragment below illustrates some of the process-oriented simulation capabilities.

```
FROM SimMod IMPORT SimTime;
      TYPE
            CombatUnit = OBJECT
                  Personnel : INTEGER;
                  Location,
                  Destination : Coordinate;
                  Speed : INTEGER;
                  ASK METHOD Status : INTEGER;
                  TELL METHOD MoveTo (IN : NewDestination : Coordinate);
      END OBJECT;
```

The first line of the example above shows the use of the IMPORT statement. The ModSim module SimMod contains the procedure SimTime. This IMPORT statement makes the DEFINITION of SimTime visible within the scope of the newly defined TYPE CombatUnit.

Also notice the addition of a new method, MoveTo, which is a *TELL METHOD*. The operation of TELL METHODS in a ModSim program differs from that of ASK METHODS. ASK METHODS perform like procedure calls in most programming languages. When an ASK METHOD is encountered, the program waits for the ASK to complete and then the next statement is executed. However, when a TELL METHOD is encountered, the program does not wait for it to complete—the next statement is executed immediately.

[9] P. Bratley, B. Fox, and L. Schrage, *A Guide to Simulation* (Springer-Verlag, 1983).

Use of the TELL METHOD combined with the *WAIT statement* causes simulation time to pass. WAIT statements can only appear in TELL METHODS and TELL METHODS can contain any number of WAITs. The following is an example of a WAIT statement:

```
WAIT DURATION real-valued-expression
        statement sequence
[ ON INTERRUPT
        statement sequence ]
END WAIT;
```

When this form of the WAIT statement is encountered, the statements after the WAIT will be executed when the specified simulation time has passed. An optional INTERRUPT clause is provided to permit other objects to stop the WAIT. If the WAIT is interrupted, the statements after the INTERRUPT will be executed.

To continue this example, the TELL METHOD, MoveTo, would contain a WAIT DURATION statement in which the time to wait is calculated based on the IN parameter Coordinate. Processes can also be combined to operate synchronously through the use of another form of the WAIT statement, as shown below:

```
WAIT FOR object TO tell-method( parameter )
        statement sequence
[ ON INTERRUPT
        statement sequence]
END WAIT;
```

Through use of this form of the WAIT, objects can synchronize their activities. This permits the direct expression of logical and physical time dependencies in a natural manner. From the example object declared above, the user could have ArmorBattalion issuing orders to its ArmorPlatoon object:

```
WAIT FOR ArmorPlatoon TO MoveTo( 2435 )
.
.
.
```

The effect of this statement within a TELL METHOD of ArmorBattalion is to synchronize its activities with those of ArmorPlatoon.

As previously mentioned, WAIT statements provide for elapsing simulated time and can be used to synchronize activities, but there are situations where processes depend on the occurrence of specific conditions. ModSim provides the class *TriggerObj*, which is used with the WAIT to allow a method to wait for some arbitrary conditions to be met:

```
WAIT FOR trigger-object TO Fire
.
.
.
```

Trigger objects contain a method—Trigger—that is invoked (fired) by some other method. Its firing permits all methods that are waiting on it to continue.

There are two constructs provided by ModSim to stop methods that have invoked WAITs: the Interrupt procedure and the TERMINATE statement. The Interrupt procedure is called from another method to stop a WAIT statement and invoke its *ON INTERRUPT* clause. For example, if this ArmorPlatoon's MoveTo method had an ON INTERRUPT clause in its WAIT: Interrupt (ArmorPlatoon, "MoveTo") would cause the statements in that clause to execute. The TERMINATE statement is called from within a process object's method to stop execution immediately.

Implementation

The ModSim language compiler is implemented as a translator. This translator generates C language source code from ModSim source code. This approach has the advantage of easy porting to other machines. The C code is compiled by the host's native C compiler to object code and then linked to produce an executable program. The needed run-time support libraries are supplied in object form with the ModSim compiler. A compilation manager is also supplied to simplify the task of compilation. ModSim is currently available for DOS*-compatible microcomputers, Sun 3, and Sun 4 computers.

Because the ModSim compiler generates C source code, it is straightforward to interface special-purpose libraries written in C. The researchers have written graphics interface, string handling, database interface, and mathematics libraries in C for ModSim.

* The widely-used disk operating system for IBM-compatible personal computers.

3 THE FSTAM OBJECT-ORIENTED COMBAT SIMULATION

Military forces may encounter a wide variety of circumstances when called into combat. Long-held assumptions about the location of probable theaters of battle may be invalid. The probability of unexpected developments highlights the necessity of flexibility as a guiding principle in designing combat simulations.[10] While it is impossible for a single simulation to account for every possible situation, simulations must be designed to allow easy modification. For example, to give detailed terrain representation and effect, it would be best to have separate terrain modules representing different terrain types, such as Eastern Europe and the Middle East. Since the type of tactics employed in different situations also differs, it is best to decouple decisionmaking from the rest of the model, allowing for different modules corresponding to different situations.

An analogous observation is valid in the context of software engineering in general: a given program may be called on to perform functions originally unanticipated by its developers. This fact has motivated software engineers to search for methodologies that would make program maintenance and modification more manageable.[11] Since computer-based simulations are software, the assertion that simulations must be made more flexible could be restated to say that simulations must fully exploit modem software engineering technology to be most applicable to today's dynamic world situation.

A characteristic of combat modeling that sets it apart from other types of software projects is the requirement that the simulation be able to model complex human decisionmaking processes within the context of battle command and control. For a simulation to be flexible, there must also be provisions for analyzing the decisions represented by simulation events and modifying the decisionmaking process to account for changes in either the environment of the simulation or changes in combat doctrine.

This chapter describes the design of the FSTAM 2.0 combat simulation using the ModSim language. A goal of the design is the ability to specify decisionmaking parameters in such a way that they will be flexible and accessible.

Two major objectives of FSTAM 2.0 are (1) to develop a combat model that represents engineers at a sufficient level of detail to permit its use as an analysis tool for studying engineer force structure and effects, and (2) to investigate simulation technologies and methodologies, and to design and implement a flexible, object-oriented architecture for combat simulations.

Combat Simulation Design

The high-level design of the simulation interface is based on the Model-View-Controller paradigm pioneered in the Smalltalk object-oriented environment.[12] The Model contains the underlying applications program, the View is the way the Model is presented to the user, and the Controller determines the way the user interacts with the Model. In the case of FSTAM, the Model comprises the essence of the combat simulation; it contains no user input/output other than messages received from and

[10] J.F. Dunnigan and A. Bay, *A Quick and Dirty Guide to War* (William Morrow, 1982).
[11] R.E. Fairley, *Software Engineering Concepts* (McGraw-Hill Book Co., 1985).
[12] G.E. Krasner and S.T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *Journal of Object-Oriented Programming* (August/September 1988), pp 26-49.

sent to the Controller, which in turn sends and receives messages to and from the View(s). Figure 1 illustrates the subdivision of component objects into these groups.

The primary goal of this design is to allow flexibility in terrain effects, decisionmaking, attrition calculation, and engineer task scheduling. Additionally, the design allows the representation of hierarchical command relationships and a realistic model of communications networks. This is accomplished by having the combat units contain no more than the stated variables, including leader and subordinate relationships and methods for high-level behaviors. All other objects, as indicated in Figure 1, are related to manipulation of the unit objects. For example, if a unit moves from one place to another it informs the map of its desired location. The map handles the specifics of unit movement, such as timing and fuel consumption. It reports changes to the unit such as its location or when the unit discovers something in the environment (the map knows the location of all units). In a similar way, orders and their subsequent decomposition into high-level behaviors are propagated throughout the chain of command via Communications Net objects and Interpreter objects. The Map and Battle Manager combine to help the units fight, and the Map and the Task Manager help the engineer units schedule and execute work orders. Thus all the major activities in the simulation are independent, illustrating how the object-oriented paradigm supports flexibility and maintainability.

Model evolution proceeds naturally through the use of inheritance. A new object can be added to the model by deriving it from an existing object without making changes in the rest of the program. Moreover, due to the modularity imposed on the programmer, the interaction between objects is clear and localized so changes in the design of component objects can be made with fewer complications. The class hierarchy allows abstraction of objects and behaviors; this provides for modeling of specific details by inheriting an abstract class and making changes to represent the particular details.

Model Overview

When designing a combat simulation there are several factors that influence what events (as opposed to *how* events) will be simulated. A major goal of the FSTAM simulation is a detailed representation of engineer activities. This section discusses these considerations and how they influence the model.

Resolution

A major decision for the development of a combat model is the choice of resolution or scale. As the goal is to represent engineer activities in detail, and since engineers work independently at platoon and company level, it was decided to represent combat units at the company level and engineer units at both platoon and company levels. This scale permits realistic modeling of the interaction between combat units and engineer units. The effects of an individual engineer unit will be noticeable by company-level combat units.[13]

Terrain

The familiar hexagon terrain map is used. Terrain variation is represented by a set of types (urban, forest, mountain, etc.), and each type affects trafficability and line of sight. Roads and rivers run from

[13] J.E. Snellen, *The MALOS Combat Engineer Simulation Environment (Version 3.0)*, Technical Report (TR) P-86/02 (U.S. Army Construction Engineering Research Laboratory, August 1986).

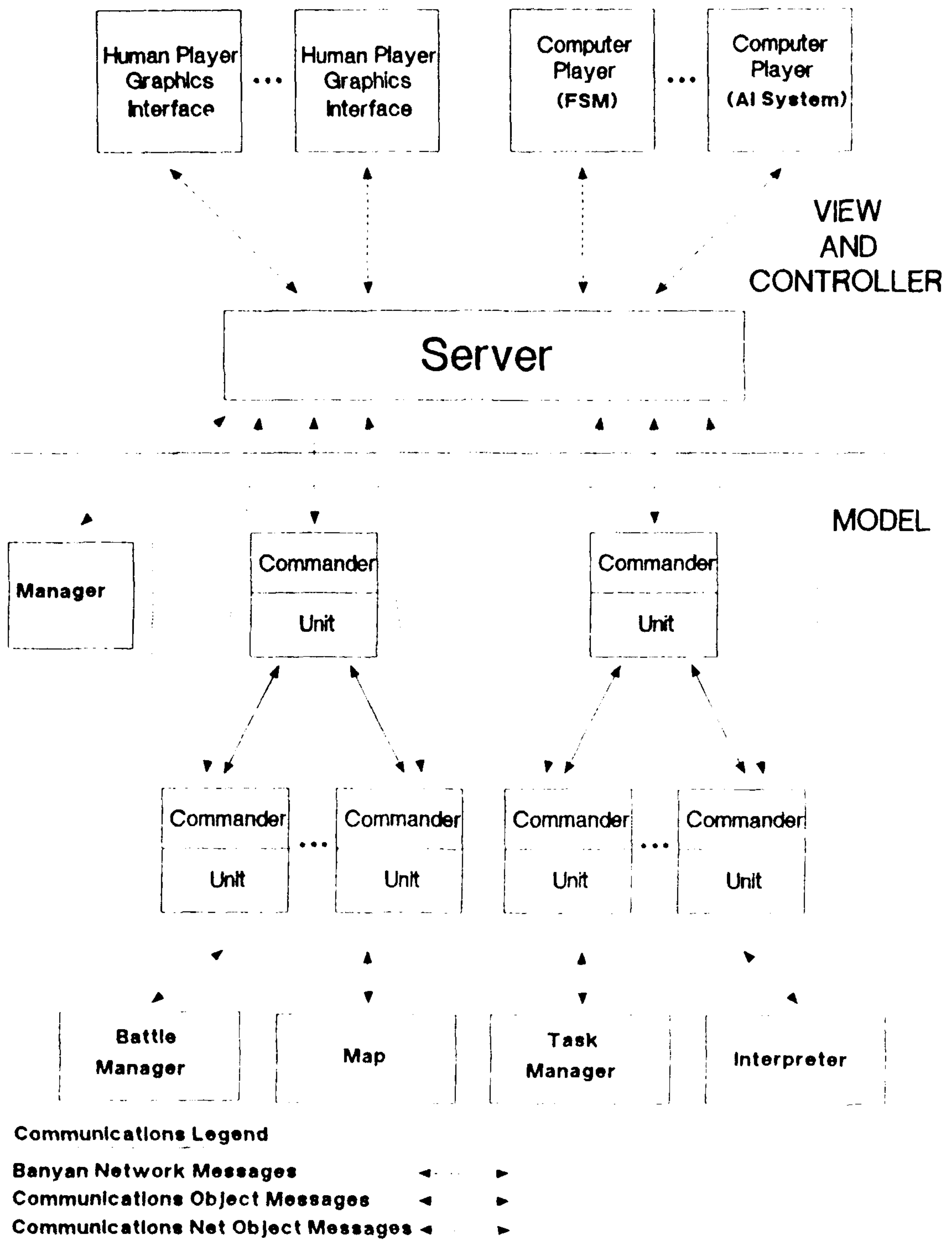


Figure 1. FSTAM system architecture.

hexagon center to hexagon center. Movement is based on a set of road types (combat trail, highway, etc.). There are also named objectives and control lines. To represent combat units at company level, the diameter of a hexagon represents approximately 200 metres.

Attrition

An important consequence of choosing this scale is that a discrete probability-of-hit/probability-of-kill type of attrition algorithm can be effectively used. This approach can be restricted so only integer values and variables are required. This eases the problems involved with portability and the ability to reproduce results from saved data. Another factor in choosing this type of attrition calculation is that the underlying parameters are readily available. The probability that one weapon system will hit or kill another weapon system, along with various modifiers reflecting the effects of smoke, weather, and cover, has been tabulated and recorded.

Although units are represented at the company level, attrition is calculated at the platoon level. Each weapon system locates a target for each round of combat if a line of sight can be established. The weapon system first determines a clear-field-of-fire-probability-of-hit based on range by table lookup. This yields an integral percentage between 0 and 100. Then the various modifiers, such as smoke, weather, level of daylight or darkness, target and firer movement, and target and firer damage, are applied to this base probability. A call is then made to a pseudorandom number generator that returns an integer between 0 and 100. If the number is less than the modified probability, a hit occurs and hits are tallied for use in the next round of fire. If a hit occurs then the probability of a kill at that range is determined; the pseudorandom number generator is called again and the appropriate outcome is recorded.

Engineer Effects

One result of the decision to scale at company level is that engineer activities are modeled at both platoon and company level. If combat units were larger, the level of visible engineer detail would diminish. Attrition would necessarily be calculated by a fuzzier algorithm, such as a Lanchester-type model,[14] or possibly a discrete method in which very little detail would be visible. Due to computer size and the chosen scale, the largest unit representable is the division. Although this scale does not provide for strategic considerations, it is large enough to see the effects of planning.

System Architecture

As stated previously, the design of the FSTAM simulation is based on a Model-View-Controller paradigm. In the Model-View-Controller approach, a primary design goal is to develop loosely coupled objects so one part of the simulation can change without significantly disrupting the rest of the simulation. This loose coupling is realized through the use of a message-passing paradigm. In the object-oriented programming community, the word "message" usually describes a form of function invocation. In this section the word message is used in a more generic sense—to denote a small packet of information. Through careful design of the object-to-object interfaces, one object can make a request of another object without the former needing to know the argument list of the function required to elicit the desired behavior. The base objects in the model are structured around the concept of passing *message* objects.

[14] J.G. Taylor, *Lanchester Models of Warfare* (Military Applications Section, Operations Research Society of America, 1983).

The use of messages facilitates loose coupling, as the sender is informing some other object in the model of a change in the state of the simulation or requesting something from another object. How the receiver responds to the message is immaterial to the sender. The *message* object's data fields are the identification number of the sender and receiver, a code indicating the type of message, a priority of transmission, the time the message was sent, the time it was received, and an encoded character string that comprises the text of the message. The text of the messages is in a format defined by the FSTAM language. An *interpreter* object class is provided to decode the FSTAM language. A particular class of objects passes the message objects back and forth. These message-passing objects control the simulation and model battlefield communications.

The View and Controller portions of the simulation are contained in a separate program referred to as the *Gamer*. There can be as many copies of the Gamer program running as there are units in the simulation. The Gamer and the Model communicate with each other via the *server* object, which uses Banyan local area network protocols.

The complete class hierarchy is shown in Figure 2.

Nonmessage Passing Objects

Interpreter Object. Messages transmitted between objects and the Gamers at this point take the form of a rudimentary command language.[15] The interpreter object's task is to encode and decode this language for the senders and recipients so the desired actions are taken. Once again, this interface to the rest of the model is entirely generic, and a more sophisticated language and language processing capability could be installed. The declaration for this object is:

```
InterpreterObj = OBJECT

        ASK METHOD Interpret (IN command : ARRAY OF CHAR;
                        IN length : INTEGER;
                        IN commander : CommanderObj);
END OBJECT;
```

Since this object only has one purpose, it only has one method. The purpose of this method is to decode an external message and to initiate the commander object's appropriate method. This in turn leads to some action in the unit associated with the commander.

Server Object. The *server* object functions as the central switchboard for the entire model. It maintains the mapping between commander object/unit object pairs and their source of decision logic, and manages the sending and receiving of the messages between them. The messages between the server object and the decision logic sources use a Banyan network and its protocols. The declaration for this object is:

```
ServerObj = OBJECT

        PortAddr : IPCPort;
        Socket:    INTEGER;
        BanVect:   INTEGER;
```

[15] C. Subick and M.A. Fiddes.

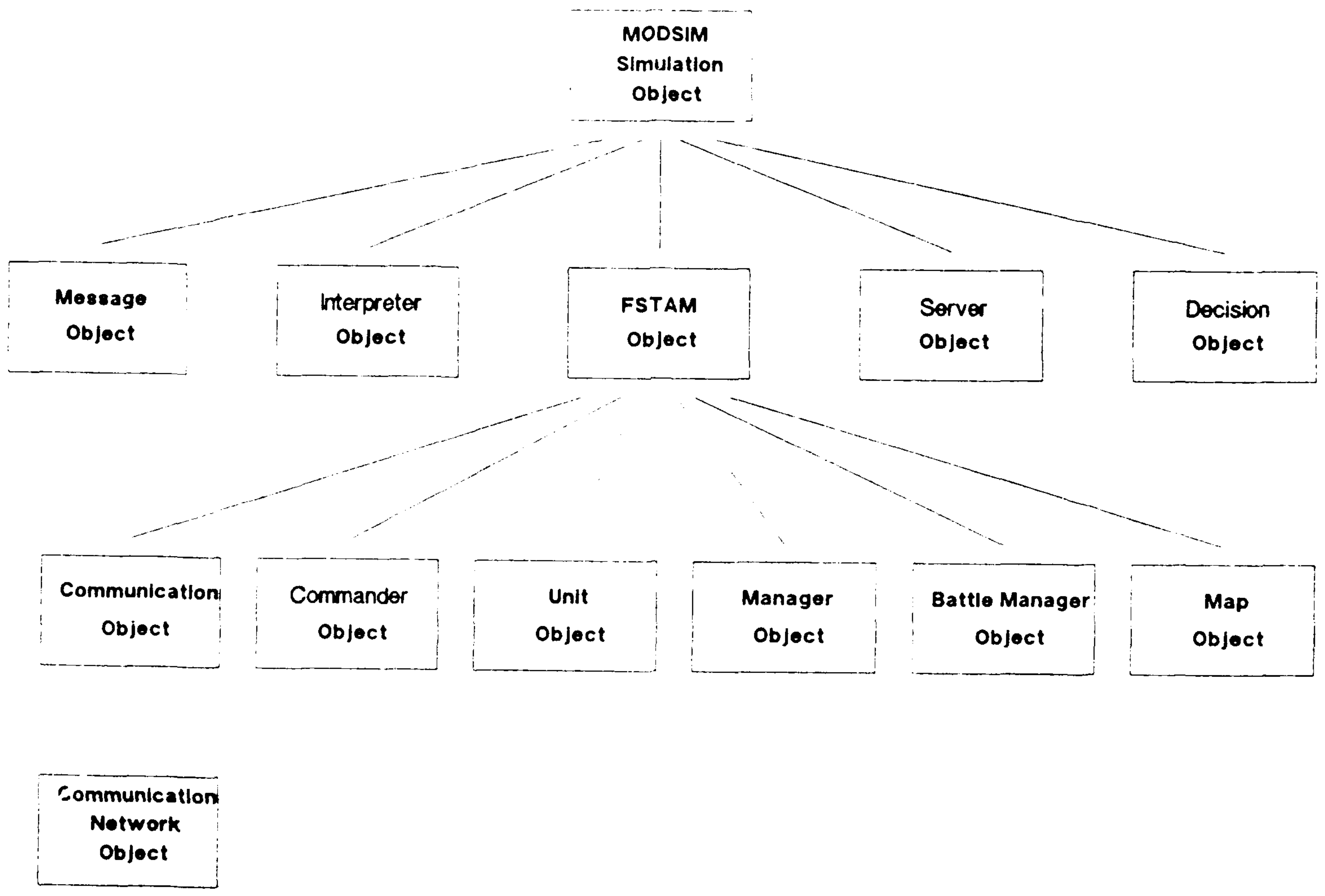


Figure 2. FSTAM class hierarchy.

```
    RecAddr:  IPCPort;
    RecBuf:   MesRec;
    SndBuf:   MesRec;
    SBoard:   ARRAY[MinPossessable..MaxPossessable] OF SwtchType;
    PendList: PListType;

    ASK METHOD ServerSend( IN fid: FSTAMID; IN type: INTEGER; IN
          buflen: INTEGER; IN buf: ARRAY OF CHAR);
    ASK METHOD CheckStatus;

    PRIVATE

    ASK METHOD ObjInit;
    ASK METHOD ObjTerminate;
    ASK METHOD DeliverMessagesFor( IN id: FSTAMID);
    ASK METHOD DeliverEACK( IN id: FSTAMID);
    ASK METHOD DeliverXACK( IN id: FSTAMID);
    ASK METHOD UpdateLastMessageTime( IN id: FSTAMID);
    ASK METHOD ExpungeMessagesFor( IN id: FSTAMID);

END OBJECT;
```

The most important features of this object are the fields SBoard and PendList, and the method ServerSend. *Sboard*, which is an array of *SwtchType*, contains the information linking commander objects and human or computer players. *PendList* is a list of messages from human or computer players to their commanders that will be distributed to commanders and run through the interpreter. *ServerSend* is responsible for sending information about the new state of the simulation as a result of external commands or requests. The remaining fields and methods assist in executing these functions.

Message-Passing Objects

The message-passing objects are called *FSTAM* objects (FSTAMObj). They comprise the major functional pieces of the model: the *controller*, the *unit*, the *commander*, the *manager*, the *map*, and the *battle manager* objects. The majority of the interaction between these objects pertaining to control of the simulation is done via messages. These objects send and receive messages to and from each other by way of the *communication* object and its derivative, the *communication network* object.

The declaration for the FSTAMObj is:

```
FSTAMObj = OBJECT

    ID :       FSTAMID;
    FSTAMType: FSTAMTypes;

    ASK METHOD SendMessage( IN message: MessageObj);
    ASK METHOD ReceiveMessage( IN message: MessageObj);
    ASK METHOD Setup( IN newID : INTEGER);
    ASK METHOD OutputState();
    ASK METHOD Halt();
```

```
        PRIVATE

        Messages : MessageListObj;

    END OBJECT;
```

This defines the base message-passing object, which has methods and fields to send, receive, and store messages. The fields FSTAMID and FSTAMType are used for internal identification purposes. The method Setup is overridden in all subsequent derived classes, and its purpose is to initialize the fields of the specific object class from information stored in a database.

Communication Object. The communication object handles simulation-control messages and certain types of battlefield communications. The simulation-control messages are typically between the map, commander, manager, battle manager, and task manager objects. The communication object serves as a central collector and distributor of messages. This allows the state of the simulation to be tracked more easily than if the objects dealt with each other directly, and maintains loose coupling. Its declaration is:

```
CommunicationObj = OBJECT(FSTAMObj)

        ASK METHOD ObjTerminate();

        PRIVATE

        Delay :       INTEGER;
        Loss :        INTEGER;
        Capacity :    INTEGER;
        DeliveryList : QueueObj;

        TELL METHOD DeliverMessage();

        OVERRIDE

        ASK METHOD SendMessage( IN message: MessageObj);
        ASK METHOD ReceiveMessage( IN message: MessageObj);
        ASK METHOD Setup( IN newID: FSTAMID);
        ASK METHOD OutputState();
        ASK METHOD Halt();

END OBJECT;
```

The principal feature of this object class is a set of fields to define the communication channel parameters—message delay, loss, and channel capacity. There is also a queue to hold messages received and a tell method, which passes simulated time, to deliver the messages. Note here the first use of polymorphism, in that the methods to send and receive messages are specialized through the use of the OVERRIDE keyword.

Communication Network Object. There can be any number of communication network objects, and these are designed to model battlefield communications. Unit objects are connected to other unit objects via communication network objects in a hierarchical fashion that simulates real-life communications networks. As with the base communication objects, these serve as central manipulators of messages to maintain loose coupling between the unit objects. In addition, they model message delays and losses. The declaration for this object is:

```
CommunicationNetObj = OBJECT(CommunicationObj)

        ASK METHOD AddToNet (IN newmember : FSTAMID);
        ASK METHOD RemoveFromNet( IN member : FSTAMID);

        PRIVATE

        memberlist : ListObj;

        OVERRIDE

        ASK METHOD SendMessage( IN message: MessageObj);
        ASK METHOD Setup( IN newID : FSTAMID);
        ASK METHOD OutputState();
        ASK METHOD ObjTerminate();

END OBJECT;
```

The methods AddToNet and RemoveFromNet and the field memberlist provide the mechanism through which hierarchical organization is achieved. As before, most of the basic methods defined for the *FSTAMObj* have been derived and specialized.

Unit Object. The unit object represents combat and support units. These objects contain weapon systems, supplies, personnel, and equipment. To maintain maximum modeling flexibility, most of the decisionmaking that reflects doctrine and interprets orders resides in a separate module. This is explained in detail below. The only routines that reside in the unit module are movement, attrition, and message processing. Unit objects are descendants of the FSTAM object, thus they inherit communications capabilities. This permits units to be organized hierarchically to represent command and support relationships. Communication networks are established to model real-world communication systems.

```
UnitObj = OBJECT

        Side :          WhichSide;
        LocX :           COORDINATE;
        LocY :           COORDINATE;
        ToX :           COORDINATE;
        ToY :           COORDINATE;
        Commander :     FSTAMID;
        Comm :            FSTAMID;
        Dead :          BOOLEAN;
                Moving :          BOOLEAN;
        Firing :        BOOLEAN;
        WeaponsSystem : WeaponsSystemType;
        SightsInUse :   PBoolean;
        SightDefense :  DArray;
        NumberSystems : INTEGER;
        Sees :          PBoolean;

    ASK METHOD Attrit( IN Loss: INTEGER);
        ASK METHOD MoveAStep();
        ASK METHOD ChangePosition( IN Xcoord, Ycoord: COORDINATE);
        ASK METHOD MoveTo( IN targetX, targetY: COORDINATE);
    ASK METHOD FireAt( IN targetID: FSTAMID);
    ASK METHOD SetComm( IN comm: FSTAMID);
    ASK METHOD SetSightsInUse( IN element: INTEGER;
                    IN value: BOOLEAN);
    ASK METHOD SetSees( IN unit: FSTAMID; IN value: BOOLEAN);
    ASK METHOD ChangeSees( IN unit: FSTAMID; IN value: BOOLEAN);
    ASK METHOD ObjTerminate();

    OVERRIDE

    ASK METHOD SendMessage( IN message: MessageObj);
    ASK METHOD ReceiveMessage( IN message: MessageObj);
    ASK METHOD OutputState();
    ASK METHOD Setup( IN newID: FSTAMID);

END OBJECT;
```

This is a basic unit template. Fields provide a certain amount of generic information and methods provide generic behaviors that one would expect a combat unit to have. More specific units, such as mechanized or engineer units, would be derived from this object class and contain more fields and methods pertinent to their specific functions.

Commander Object. There is a commander object attached to each unit object in the simulation. The commander object determines responses to messages received by the unit objects after they have been decoded by the interpreter object. For example, if a unit received an order to attack an objective, the commander would invoke the unit's MoveTo method to move it to the objective. Then, if an enemy unit was discovered, the commander would invoke the fire method to attack the enemy unit. The commander objects are attached to the server object, and can be taken over by a human or a computer player through this interface. The object declaration is:

```
CommanderObj = OBJECT(FSTAMObj)

    Possessed :   BOOLEAN;
    CommanderOf : FSTAMID;
    Side :        WhichSide;
    Level :       WhichLevel;
    Type :        WhichType;
    Name :        FSTAMName;

    ASK METHOD Possess();
    ASK METHOD Exorcise();
    ASK METHOD ServerReceive( IN buflen: INTEGER;
                    IN buf: ARRAY OF CHAR);
    ASK METHOD SetName( OUT name: ARRAY OF CHAR);

    { Methods which correspond to unit methods follow.}
    ASK METHOD MoveTo( IN targetX,targetY: COORDINATE);
    ASK METHOD FireAt( IN targetID: FSTAMID);
    ASK METHOD SetComm( IN commnet: FSTAMID);

    OVERRIDE
    ASK METHOD SendMessage( IN message: MessageObj);
    ASK METHOD ReceiveMessage( IN message: MessageObj);
    ASK METHOD Setup( IN newID: FSTAMID);
    ASK METHOD OutputState();
    ASK METHOD Halt();

END OBJECT;
```

The commander object's fields define the position it and the unit it controls occupy in the military hierarchy, and whether or not it is connected to a human player or a computer player. If Possessed is true, a human is connected to the commander; if false, a computer player is connected. The methods *Possess* and *Exorcise* control the connection and disconnection procedure. The remaining methods deal with controlling the associated unit object and message passing.

Manager Object. The manager object's task is to perform basic simulation system tasks. These include saving the state of the simulation, the restoration of a simulation from a previously saved state, temporarily suspending the simulation, and stopping the simulation. The object declaration is:

```
ManagerObj = OBJECT(FSTAMObj)

    Commander : FSTAMID;

    ASK METHOD Stop();
    ASK METHOD Save();
    ASK METHOD Restore();

    OVERRIDE

    ASK METHOD ReceiveMessage( IN message: MessageObj);
    ASK METHOD Setup( IN newID: FSTAMID);
    ASK METHOD Halt();

END OBJECT;
```

Map Object. The map object contains all information about the displayed terrain. The map's other capabilities pertain to managing the location of units, and, in particular, making the terrain representation invisible to the units. When a unit knows where it needs to go, it makes a request to the map object to move and supplies generic movement characteristics necessary to realistically represent the change in location. The map doesn't need to know what kind of unit is moving, and the unit doesn't need to know how terrain is represented. The object declaration is:

```
MapObj = OBJECT

    ASK METHOD Range( IN x1,y1,x2,y2: COORDINATE):COORDINATE;
    ASK METHOD RequestMove( IN unit: FSTAMID;
                IN newX,newY: COORDINATE);
    ASK METHOD RemoveRequests( IN unit: FSTAMID);
    ASK METHOD AddUnit( IN unit: FSTAMID;
              IN newX,newY: COORDINATE);
    ASK METHOD RemoveUnit( IN unit: FSTAMID);
    ASK METHOD ObjTerminate();
    PRIVATE

    NumControlLines : INTEGER;
    MapPtr :          POINTER TO MapRec;
    Units :           ARRAY[MinUnitID..MaxUnitID] OF UnitRecord;
    ControlLines :    ARRAY[1..MaxCntrlLines] OF CntrlLineRecord;
    RequestList :     ListObj;
    DODiscovery :     BOOLEAN;

    TELL METHOD ExecuteMoves();
    ASK METHOD AddToMap(IN unit: FSTAMID; IN X,Y: COORDINATE);
    ASK METHOD RemoveFromMap( IN unit: FSTAMID);
    ASK METHOD Discovery();

    OVERRIDE

    ASK METHOD OutputState();
    ASK METHOD Setup( IN newID: FSTAMID);
    ASK METHOD Halt();

END OBJECT;
```

Most methods deal with moving units around and in general keeping track of them, such as *Discovery*. The primary data fields are *MapPtr* and the *Units ARRAY*. MapPtr is a pointer to an array that contains the terrain information, including roads, rivers, and obstacles. The Units array contains the global information on all units.

Battle Manager Object. The purpose of the battle manager object is to resolve rounds of fire between units so they do not need to know the details of the attrition calculation. The firing unit acquires a target and makes a firing request to the battle manager. This request is placed on a list and executed. The battle manager then interrogates the map for the range and other terrain or environmental modifiers. It then determines the unit movement and damage modifiers, applies these to the probability of hit, and then proceeds as previously described. This approach decouples terrain representation from the units and compartmentalizes the attrition methodology and calculations. This in turn minimizes the effect of changes to any particular module on other modules in this fundamental part of any combat simulation. The declaration is:

```
BattleManagerObj = OBJECT(FSTAMObj);

    ASK METHOD RequestFire( IN unit,target: FSTAMID);
    ASK METHOD RemoveRequests( IN unit: FSTAMID);
    ASK METHOD ObjTerminate();

    PRIVATE

    RequestList :     ListObj;

    TELL METHOD ResolveFire();

    OVERRIDE
    ASK METHOD OutputState();
    ASK METHOD Setup( IN newID: FSTAMID);
    ASK METHOD Halt();

END OBJECT;
```

Task Manager Object. The task manager object's function is to calculate the required resources and execution times for a given engineering task. This object has routines available to the engineer headquarters for forecasting and planning purposes. Once a request has been matched to an engineer unit and the unit is in position to execute the task, the task manager coordinates all aspects of the execution of the request. This includes all the requisite bookkeeping of assets required and expended. The declaration is:

```
TaskManagerObj = OBJECT(FSTAMObj);

    ASK METHOD RequestJob( IN job: JobType; IN X,Y: COORDINATE);
    ASK METHOD RemoveJob( IN job: JobType; IN X,Y,: COORDINATE);
    ASK METHOD ObjTerminate();

    PRIVATE

    RequestList :     ListObj;

    TELL METHOD PerformJob();

    OVERRIDE

    ASK METHOD OutputState();
    ASK METHOD Setup( IN newID: FSTAMID);
    ASK METHOD Halt();

END OBJECT;
```

External Rule Base and Artificial Intelligence Capabilities

When the rules or doctrine governing the modeling of specific entities are embedded in the code for the simulation, they become hard to analyze and difficult to change. For example, a combat unit's behavior in a given situation is generally determined by a complex set of conditions. In conventional modeling methodologies, the logic that governs the unit is encoded in a set of conditional statements distributed across a number of modules. Such a representation has a number of disadvantages. First, in a simulation of any size, it becomes difficult to identify all the places where a unit's behavior may be influenced. Second, the logic that captures the decisionmaking process is intermixed with the logic that implements the decision (e.g., deciding to move and moving). Finally, the behavior is hard-coded and cannot be changed without rebuilding the entire simulation.

Despite the success of discrete-event simulation and its increasing use in a number of different application areas, current techniques have proved largely inadequate for modeling complex systems that attempt to include some element of human decisionmaking. In real combat situations, decisions are often based on questionable or incomplete information. Historically, simulation techniques have been most effective when the question of what to do next at any given point in the simulation could be reduced to a relatively simple formula and easily embedded in the simulation. For example, in simulating a queue, the decision of which waiting job to process might be reduced to "always select the item with the shortest processing time." While such a formulation might be adequate for some analyses, it fails to capture the decisionmaking process that would occur in a real combat setting. In combat, a commander might have to consider numerous pieces of information, possibly including conflicting subgoals, when formulating a plan that would culminate in selecting a course of action. Experience has shown that the ability to make complex decisions based on the state of the system plus a plan of action is difficult, if not impossible, using conventional simulation methods and tools.

To address these limitations, the proposed architecture includes a rule-based approach to unit behavior that separates the logic of decisionmaking from the rest of the simulation, and encapsulates that logic in a cohesive and accessible form. The behavior of each simulation entity, such as a combat unit, is controlled by a decisionmaking agent. This agent is indirectly attached to the unit object through the unit object's commander object and the server object. Under this arrangement, the unit object contains the logic for responding to decisions (e.g., executing orders). Since the unit is separated from the source of the decisions by the commander and server, it is possible to interchange the source of the decisionmaking (human players, hard-coded scripts, simple programs, expert systems, or finite-state machines) without disturbing the unit or the rest of the simulation (see Figure 1). Furthermore, the loose coupling provided by the message-passing paradigm allows the decisionmaking agent to be defined and to execute in a distributed fashion external to the rest of the simulation, communicating the results of its reasoning through the message-passing protocols.

This design will allow research into the development of intelligent agents that will serve as autonomous decisionmaking entities when units are under computer control. The properties and architecture of intelligent agents are part of the discipline of artificial intelligence (AI) and have received much attention.[16] The work of defining these agents for use in combat simulations remains an area of active interest that will be pursued in future research.[17]

Configuration

The simulation requires a minimum of three IBM AT-compatible machines connected by a Banyan network. One machine runs the simulation and functions as a server. A minimum of two players, whose machines run the gamer programs, are required. These players fill the commanding roles of opposing sides. However, the architecture is designed so each unit may have either a human or computer player. Therefore, in its fullest form, the model will be able to support the same number of human players as there are units in the game. It will also be possible to let the computer play some or all of the units. Finally, an additional player can serve as manager, controlling the system functions to suspend, save, or restore simulations.

[16] M.R. Genesereth and N.J. Nilson, *Logical Foundations of Artificial Intelligence* (Morgan-Kaufmann Publishers Inc., 1988).
[17] L.E. Widman and K.A. Loparo, "Artificial Intelligence, Simulation and Modeling: A Critical Survey," *Artificial Intelligence, Simulation and Modeling*, L.E. Widman, K.A. Loparo, N.R. Nielson, eds. (John Wiley and Sons, 1989).

4 HYPERTEXT APPLICATIONS TO MODELING AND SIMULATION

Information Management Problems of Simulations

The design, construction, operation and maintenance of simulations is a complex activity that generates much disparate documentation, data, and source code. A recent General Accounting Office (GAO) report[18] on DOD simulations points to the vital need for organizational support structures, documentation, and reporting if models are to remain credible. Software engineering has developed over the past 20 years as a discipline capable of managing this complexity.

Traditional software engineering environments have focused on the development and integration of tools such as editors and compilers, mainly in support of code production. More recently, computer-aided software engineering (CASE) tools have focused on the analysis and design phases of software development. These efforts are sorely needed to improve software quality and productivity. The support required for large simulation models, however, goes far beyond the software development and maintenance phases of the product life cycle. Tools and techniques to capture, in a well structured manner, all related information generated during the process of model development are required.

This chapter discusses some first steps toward using *hypertext* as the data model for a simulation support environment. The FSTAM simulation was used as a test for the application of hypertext to an existing simulation. Some general requirements for a hypertext-based environment for simulations are suggested.

Characteristics of Hypertext

Hypertext is a method for storing and accessing information in a manner resembling human thought processes. Unlike books or word processors, which portray documents as linear sequences of sentences and paragraphs, hypertext systems permit explicit referential links between concepts. This feature allows users to travel through the online document in a nonlinear manner, following their own train of thought. Each concept in a hypertext system is called a *node*. Nodes are not restricted to textual information only: they may be graphic images, sound, and even video. The relationships between nodes are called *links*. Thus, an online hypertext document can be thought of as a network containing both information (nodes) and the relationships (links) among the information. Hypertext systems provide features for structuring and traversing these networks.

The term "hypertext" is credited to Nelson,[19] who first explored its uses in the late 1960s. There is renewed interest in hypertext, and many hypertext systems are now available. Conklin, in his survey of hypertext systems, states:

> Hypertext invites the writer to modularize ideas into units in a way that allows (1) an individual idea to be referenced elsewhere, and (2) alternative successors of a unit to be offered to the reader.[20]

[18] *DOD Simulations: Improved Assessment Procedures Would Increase the Credibility of Results*, GAO Program Evaluation and Methodology Division (PEMD) Publication 88-33 (GAO, December 1987), pp 47-53.

[19] T.H. Nelson, "Getting it Out of Our System," *Information Retrieval: A Critical Review* (Thompson Books, 1967), pp 191-120.

[20] E.J. Conklin, "Hypertext: An Introduction and Survey," *IEEE Computer*, Vol 2, No. 9 (1987), pp 17-41.

This description of hypertext is strongly reminiscent of the software engineering concepts of modularity, coupling, and cohesion. Experience shows that a hypertext representation of software systems—particularly models—can lead to better understanding and maintenance. Additionally, hypertext provides a framework for organizing all information related to a model into a logical structure.

General Requirements

Support for System Life Cycle

A major requirement for a hypertext-based simulation environment is support of the system life cycle. There are many life-cycle models of software development, including the phased life cycle, rapid prototyping, and evolutionary development. The hypertext system must be robust enough to sustain any of these and their variants. Two life-cycle models are considered here.

Nance has formulated the Conical Methodology specifically for model development.[21] This methodology addresses the total model development process, of which software development is but a part. The rationale for the Conical Methodology is based on the following premises:

1. Model specification and model documentation should be accomplished within the same activity.
2. A hierarchical structure for model development is advantageous.
3. The clarification of nomenclature is a necessary precursor to documentation standards.

Clearly, hypertext can facilitate the accomplishment of the first principle listed above by providing a common environment for both specification and documentation. Inherent to hypertext is a hierarchical data model that can represent both top-down analysis and bottom-up development. Also, the implementation of a hypertext system necessitates the clarification of nomenclature and documentation standards.

The phased life-cycle model of Department of Defense Standard 2167 (DOD-STD-2167)[22] is shown in Figure 3. This life cycle addresses the software development activity within the model development process. Full implementation of DOD-STD-2167 is not necessary, but it provides a sufficient and traditional set of documents for the initial hypertext network. As a minimum, the following phases must be fully supported:

— Planning
— Requirements analysis
— Unit testing
— Component integration
— Component testing
— System testing
— Validation
— Verification
— Specification
— Design
— Unit coding.

[21] R.E. Nance, *The Conical Methodology: A Framework for Simulation Model Development*, Technical Report SRC-87-002 (System Research Center, Virginia Tech, 1986).
[22] DOD-STD-2167, *Defense System Software Development* (DOD, June 1985), pp 2-3.

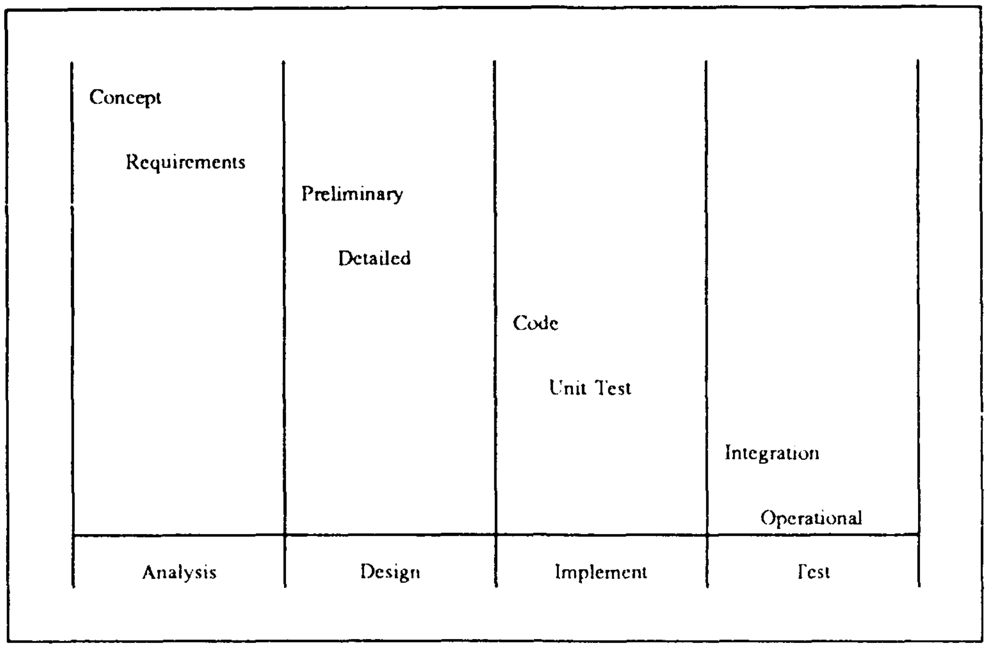


Figure 3. The phased life cycle for software.

An advantage of using the DOD development model is the inherent provision for configuration management. Configuration management introduces the important concepts of *baseline*, *configuration item*, and *version control*.[23] A major criticism of DOD-STD-2167 and configuration management is the large number of documents produced. Hypertext can provide a framework to integrate the desirable features of the phased life-cycle model and configuration management into a usable system.

The above requirements provide the conceptual topology for a hypertext system coordinating all of the documentation, input data, output data, source code, and object code for a simulation. Upon initialization of such a system, the topology specifies the embryonic hypertext network from which the system will evolve. The nodes of the system are templates of appropriate format for the type of document, database, or other structure. The initial links are those chosen by the designers as necessary to provide rudimentary connections among the template nodes. Users involved at each phase and level of the project will tailor the network to meet their needs. Project managers will exercise access control over the network, ensuring consistency with project requirements.

The hypertext system must be built with recognition of software standards that permit interfacing with a variety of other software packages. These packages are important to both developers and users in all phases of the system life cycle. This approach provides for implementation language and methodology independence. For example, during the analysis and design phases, CASE packages could be used. Several hypertext-based CASE tools are under development.[24] While these tools are promising, they do

[23] J.K. Buckle, *Software Configuration Management* (Macmillan Press, 1982).

[24] J. Bigelow, "Hypertext and CASE," *IEEE Software* (Institute of Electrical and Electronics Engineers [IEEE], March 1988), pp 23-27; A. Hecht, "The Hypertext Features of Teamwork Analysis and Design System," *Proceedings of the Second International Workshop on Computer Aided Software Engineering*, Vol 1 (1986), pp 1-44.

not address the total system life cycle nor do they provide an operational environment for users of the software.

User Support

Another major requirement for a hypertext-based simulation environment is user support. A major weakness of current simulations is the lack of a user support environment. The hypertext system used to develop the software should provide an environment for operating the simulation. This environment must provide for scenario construction and analysis. One benefit of this approach is that the analysis, assumptions, and comments of the developers are available and easily accessed. Information can be cross referenced to both input and output data. Thus, users can trace input data requirements through analysis, design, and implementation to output. This traceability provides great insight on how to use the model properly.

Multiuser Capabilities

Finally, any hypertext system capable of supporting a simulation environment must be a multiuser system. An essential characteristic of hypertext is the provision for cooperative work. Distributed hypertext systems will become necessary for supporting remote sites. This feature is particularly needed during the fielding and maintenance phases because developers and users may be geographically separated.

An Example Application

FSTAM consists of five main programs totaling more than 30,000 lines of FORTRAN code. To better understand the simulation, and to aid in the development of the next version, the researchers investigated the application of hypertext.

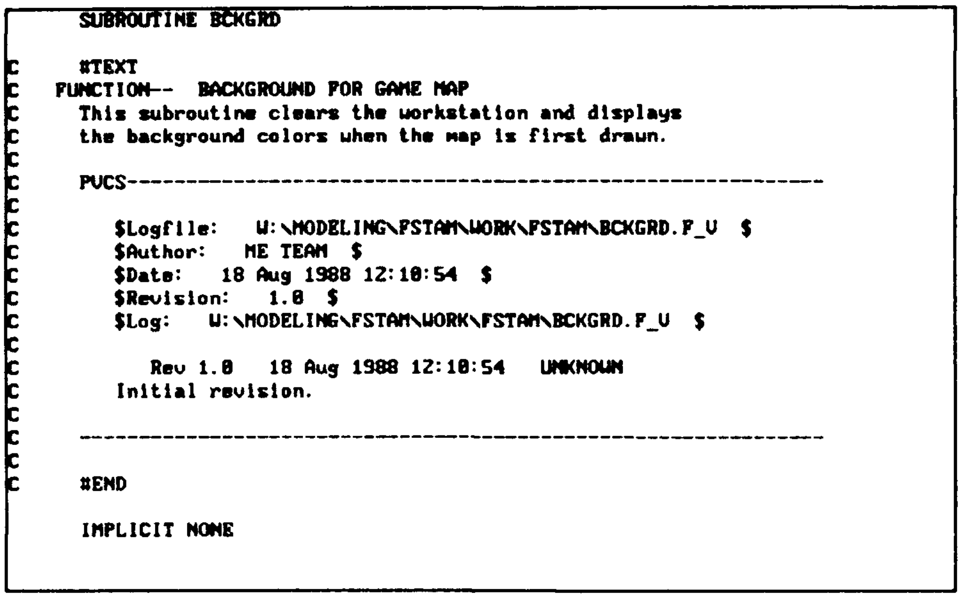
After reviewing several personal computer (PC)-based hypertext systems, *PC-Hypertext*[25] was selected—primarily because it stores files in ASCII text format. The other hypertext packages available stored files in their unique internal formats, which was unacceptable to text editors and compilers.

The first goal was to enter the source code into the hypertext system. Several ways were found to generate hypertext views of the programs. One approach is outlined here.

A desirable view of the software is that of the program's calling tree. The calling tree is a listing of each module (subroutine or function) and the modules that it calls. It is a typical output from a FORTRAN source code analyzer. This view gives one the ability to traverse the source code in the program's logical execution sequence. In hypertext, this view is defined by a network where each node is a module (consisting of a FORTRAN source code file). Each node in the network has a set of links that correspond to the FORTRAN subroutines called. A utility program was written to take the calling tree file shown in Figure 4 and transform it into the hypertext network. There are many other useful views of the code, such as the reverse calling tree. A network based on the reverse calling tree shows all modules that call a given module. When making changes to the source code, the ability to identify and quickly check these dependent modules is invaluable.

Other information included in the hypertext network are nodes for the user manual, assumptions, comments, and enhancements. Each node is itself a subnetwork, and any node can link to any other node in the system. For example, a paragraph in the user manual may refer to the source code module that implements the feature or algorithm described. That module may in turn refer to an assumption made by

[25] N. Larson, *Hyperlink* (MaxThink, Inc., 1988).



```
      SUBROUTINE BCKGRD

C     #TEXT
C   FUNCTION--  BACKGROUND FOR GAME MAP
C     This subroutine clears the workstation and displays
C     the background colors when the map is first drawn.
C
C     PVCS-------------------------------------------------------
C
C        $Logfile:   U:\MODELING\FSTAM\WORK\FSTAM\BCKGRD.F_U  $
C        $Author:   ME TEAM  $
C        $Date:   18 Aug 1988 12:10:54  $
C        $Revision:   1.0  $
C        $Log:   U:\MODELING\FSTAM\WORK\FSTAM\BCKGRD.F_U  $
C
C           Rev 1.0   18 Aug 1988 12:10:54   UNKNOWN
C        Initial revision.
C
C     -----------------------------------------------------------
C
C     #END

      IMPLICIT NONE
```

Figure 8. First page of BCKGR.FOR.

5 PERSISTENT MODSIM

Data Management in Large Simulations

Large simulations require vast amounts of data for scenario development and generate vast amounts of related data that must be analyzed. Traditional languages used for simulation (e.g., FORTRAN, SIMSCRIPT) have no inherent data model. The two options provided in these older languages for secondary storage management are: (1) writing to flat ASCII files or (2) using a language binding to an external database management system (DBMS) such as Structured Query Language (SQL). In both cases the data manager is external to the language. This arrangement is *ad hoc* at best because traditional procedural languages lack the consistency of internal data representation required for a more intimate interface to a data manager.

To address this problem, a major thrust in language design has focused on the integration of database management facilities with modern software engineering constructs. With the paradigm in design and programming shifting from structured to object-oriented approaches, this trend toward integration of database capabilities has accelerated.[26] The successful integration of database management facilities with object-oriented concepts will allow the engineering of large, complex, data-driven applications that are both maintainable and reusable. Research has suggested that this trend can greatly benefit simulation languages and applications as well.

Object-oriented methodologies evolve around decomposing the system under construction into a group of software objects that correspond to the real-world entities being modeled. These objects embody both the description of the real-word entities (in terms of values that identify the discrete states of the object over time) and the operations that the object can perform. An object that exists external to the execution of any particular program is called a *persistent* object. A persistent object may be created before the execution of a program (via an object editor or other tool), interact with the program during its execution, and survive in the database after the program has finished.

The concept of persistent objects offer a number of benefits over traditional simulation technology. Chief among these benefits is a standard interface for the development of advanced support tools for scenario development and analysis. Instead of the laborious process of defining a massive dataset for simulation of some situation, entities in the simulation inherit their behaviors and their initial state from a library of persistent objects. Only elements unique to the situation under analysis must be specified. With tools such as an advanced version of the class-hierarchy browser that allows editing, changing the parameters of the simulation becomes as easy as viewing a particular object instance and changing some or all of its instance variables. Instead of megabytes of unstructured text and numbers being written into files, the results of the simulation are captured in the database for all users to view. Specific questions and causal relations may be discovered by database queries instead of the tedious process of scanning the vast amounts of simulation output. Such an environment would represent a quantum leap forward in the state of simulation technology.

Since all aspects of the simulation could conceivably be captured, the activity list and other internal structures that manage the simulation could also be stored in the database. This opens the possibility of

[26] J. Elliot and B. Moss, "Object Orientation as Catalyst for Language Database Integration," *Object-Oriented Concepts, Databases and Applications* (ACM Press, 1989).

building tools to aid in the creation of simulations, analogous to debuggers for conventional programming languages. A simulation debugger could be created by storing the cause-and-effect relationships that trigger state changes in objects; this would allow the evolution of the state of objects in a simulation to be traced. A byproduct of storing the internal simulation structures would be the ability to arbitrarily halt a simulation and restart it at the same place, possibly with changes to some state variables.

Research indicates that a simulation language with an integrated and consistent object/data management facility could greatly contribute to the development and use of simulations. In fact, such a facility could fundamentally change the way simulations are created and used.

This chapter describes the results of the research effort to add persistence to the ModSim language. First, the requirements for persistent object support and design considerations for the development of the prototype persistent ModSim are discussed. The prototype implementation of the persistent compiler and support tools are both described. Finally, a discussion of the deficiencies of the current prototype is included, and extensions of the prototype needed for complete simulation support are proposed.

Requirements for Persistent Object Support

Development of the requirements for a persistent integrated data-management facility must begin with an examination of current object-oriented data management models. Zdonik and Maier[27] have proposed a standard for object-oriented database design. The researchers have identified requirements from their model that appear to be essential to support persistent simulations. These requirements can be grouped into three categories: database functionality, object identity, and encapsulation.

Database Functionality

Database functionality is described in terms of the features usually found in conventional database systems.

A database has some notion of data structure and a language for manipulating the data. Generally the data entities are records and groups of records. The data manipulation language provides for operations on records and files composed of records.

A database provides for the grouping of entities through relationships. In the relational model these relations are called *tables*. Relations can be named and the data manipulation language can query these relations. Other database models provide for one-many and many-many relationships, as in the hierarchical and network type databases.

An obvious goal of a database is to provide persistent and stable storage of data. Persistence means that the data is available after the process (program) that created it has terminated. Stability means there is some degree of robustness in cases of failure.

A database *schema* is a specification of all types and the names of all objects of those types. Most large database systems maintain the schema in a separate repository. Smaller systems may represent the

[27] S. Zdonik and D. Maier, *Readings in Object-Oriented Databases* (Morgan Kaufmann, 1989

schema as data within the system itself. The purpose of the schema is to permit disparate programs access to the information describing how the database is structured.

Most modern databases provide a query language that is declarative and supports associative access. *Declarative* means the user states the goal of a query and the query processor determines the process by which the data are obtained. The query language, especially in relational databases, can retrieve data based on associations (such as relations) among the data entities.

Views are interfaces to the database. Views are developed for particular classes of users and the operations they perform. Report and form management is usually provided as a separate application. Finally, a data dictionary is an extension of the schema concept and provides more information (e.g., input and output format) on the data contained in the database.

Object Identity

Object identity is a major requirement for object-oriented database systems. The identity of an object depends on the values of the state variables and is invariant for the life of the object. Ullman makes object identity the key characteristic that distinguishes object-oriented database systems.[28] Systems that base identity on the value of data variables are called *value-oriented systems*. Relational systems are value-oriented, e.g., key fields used to create indexes. The network model of database systems is object-oriented. An object-oriented database system provides for testing the identity of objects.

Encapsulation

Encapsulation refers to the usual meaning of the word in object-oriented systems. An object encapsulates both its data and the methods that operate on these data. The data model proposed is based on definition of an object that provides for encapsulation.

Design Considerations

A persistent object is one that is created in the runtime database of a program (possibly prior to execution), updated throughout its life to reflect state changes, and available for inspection after the program has terminated. The following paragraphs review design considerations based on the previously stated data-management requirements for achieving persistent objects.

The data entities of interest here are objects. The data model is a natural extension of current object definition facility. It is the data component (fields) of the object that will be stored along with the relationship of that object to other objects. Within the context of the current (compiled) ModSim it is not feasible and would serve little purpose to store and reload methods (i.e., object code).

There are several possibilities for determining which objects should persist. Perhaps the most straightforward approach would be to designate that all objects created in the simulation should be stored. Alternatively, the language could be ex. ·ded to introduce additional syntax to permit programmer control over which objects persist. A third possibility involves the creation of additional tools external to the compiler that would allow the creator of a simulation, within the context of scenario management, to designate which objects should persist. For purposes of simulation, the researchers considered it desirable

[28] J. Ullman, *Principles of Database and Knowledge Base* (Computer Science Press, 1988).

for all objects to be persistent. This is the approach taken in developing the original prototype of persistent ModSim, and it has several benefits. It simplifies design and implementation. It requires no language extensions and, therefore, no change to existing applications, and it relieves any additional programming burden that would be required with the latter two approaches.

Although objects may inherit partial descriptions and behaviors, each object created during the execution of a simulation is a unique entity. A database of objects must maintain the individuality of each object instance. Any two instances of a particular object class are unique, even if the values of every field variable in the two instances are equal. This characteristic is referred to as *object identity*.

The current prototype design does not require the addition of an explicit data manipulation language in order to create, access, and update persistent objects. This is done automatically within current support for objects, and is transparent to the programmers and users of a simulation. An extension to this approach offers the programmer explicit control over what objects are loaded from the database into the program; in this case, an extension of syntax and semantics that would support database queries would be necessary.

The structure of the runtime database reflects the hierarchical structure of modules and objects. Object classes are defined in modules. The description of an object class may inherit characteristics from other objects in the same or different modules. These relationships must be maintained by the database both to support the generation of object instances at runtime and to support the analysis of data after the simulation has completed. Instances of object classes constitute the bulk of any simulation.

In analyzing the problem of storing objects in a database, it was concluded that there are objects and object relationships common to all simulations. The structure of this information is referred to as the *static schema* because it does not change from database to database. Figure 9 shows a graphical representation of the static schema. The first object in the hierarchy of relationships is an instance of type MODULE. Modules may be related to other modules through importation (through use of the IMPORT statement). Modules contain classes, and classes contain fields and methods. Classes may also be related to other classes through inheritance. All this information is captured and stored in the static schema in the runtime database.

For this strategy to reduce the complexity of analyzing data generated by simulations, the database must be flexible; it must support multiple views of the data and the relationships among those data entities. Other features required are querying, views, and report and form management.

Prototype Implementation

This section describes the implementation of a prototype version of ModSim. The operation of the prototype can be separated into two phases: the compile-time functionality and the run-time functionality.

Compile-Time Functionality

At compile time a database is opened and the static schema described above is recorded. As previously stated, static schema instances correspond to information involving modules, classes, fields, and methods and their relationships. A corresponding entity is added to the database for every module referenced during compilation. Each definition module may contain any number of object class declarations; for each such declaration, a corresponding entry is added to the database. Object class declarations consist of fields (the instance variables of the class) and methods (the messages that instances of this class respond to). Information on both fields and methods is added to the database as they are encountered during compilation.

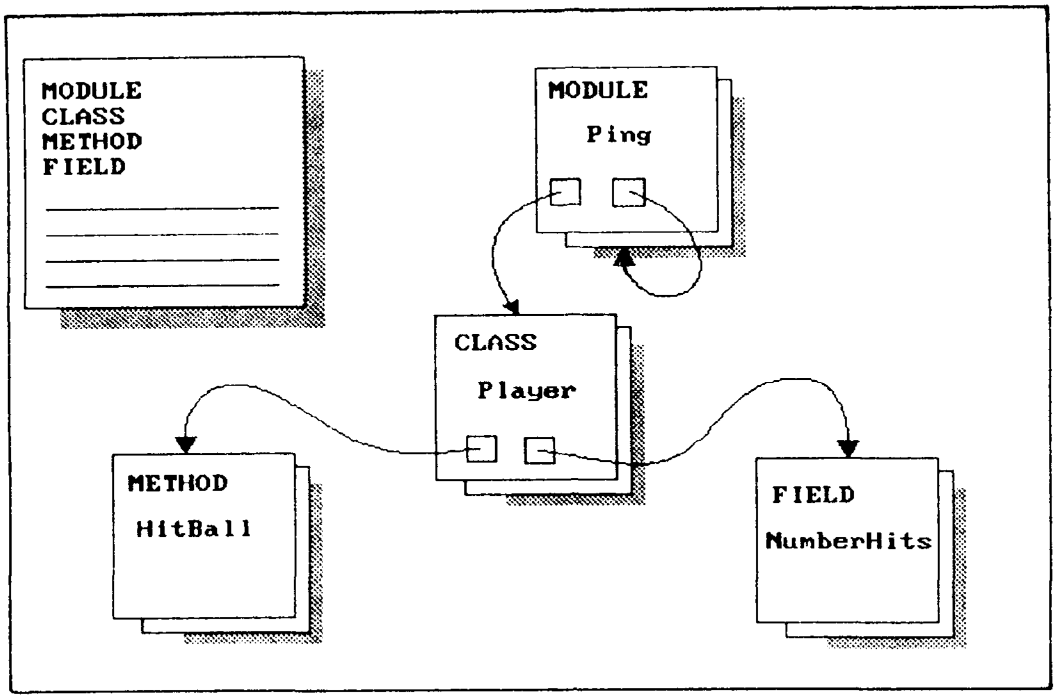


Figure 9. Graphical representation of the static schema.

As the compiler parses the source code, *dynamic schema* are generated. These dynamic schema are used at runtime to store object-class instances. Dynamic schema may be thought of as *instance templates*. These schema are called dynamic since their format differs from program to program, depending on the actual number and data types of the fields of a particular class.

In addition to the hierarchical structure of modules and classes, and the set of instance templates, additional semantic information is kept in the database. One set of information allows mapping between class names and database schema. Other information maintains the external (i.e., user-defined) names and types of the data stored in the database in addition to the data itself. This "metainformation" is useful for supporting flexible views of the database during analysis.

At the end of the compilation process, in addition to generating the code for the executable simulation, a database has been created with information about the relationships between modules, classes, and templates corresponding to each object declaration encountered in the source files. During the compilation process, the code produced by the compiler is augmented by instructions that allow the program to use the information in the database at runtime.

Run-Time Functionality

During the execution of a simulation, when an object instance is created (which must be done explicitly through the use of the NEWOBJ operator), an instance of the corresponding class template is created. It records both the initial values for field variables and the simulation time of creation. Then, as methods are invoked that alter the state of the object, updates are made to the database so the class template instance remains consistent. Since the only way to alter an object's field variables is through that object's methods, this strategy ensures the integrity of the data. Finally, when the object is destroyed, the time of destruction is recorded in the class template instance and the instance is closed.

Support Tools

This section describes the prototype class-hierarchy browser. The browser program, which is named *SeeData*, uses the persistent data model underlying the Version 1.0 compiler to provide a user-friendly interface with the persistent objects. Using SeeData, one can view the import dependencies in the module structure, the inheritance graph of a particular object, the definition of a type of object, and a particular instance of a given object. One can also switch between directories and projects in the host file system without exiting the browser.

Two example projects are provided below to illustrate both the use of the browser and its utility. The first project, called PROJECT, consists of three modules that implement a simulation of a tennis game. The second example is the FSTAM combat simulation currently under development at USACERL. This project, called FSTAM, consists of more than 35 modules.

This prototype version of the browser runs only on DOS-compatible platforms and has some stringent hardware requirements, including a virtual graphics adapter (VGA) graphics card/monitor and a Microsoft-compatible mouse. The browser checks the configuration of the host machine and will exit with an error message if the required hardware is not available.

In addition to informing the user about the hardware requirements of the program, the usage information, which is displayed if the program is invoked incorrectly, also indicates that the program may be executed without any arguments, or with an optional project name, followed by an optional version number. For example, to bring up SeeData without a preloaded project database, the user need only enter "SEEDATA" on the DOS command line. If the user wishes to view the initial version of a project named TEST, the user may enter "SEEDATA TEST" on the command line. This is equivalent to entering "SEEDATA TEST 0," for the initial (or "0th") version of the database is selected by default when a project name is provided without a version number. If the user wishes to view the results of simulation execution he must specify the version of the database he wishes to view. Each time a persistent program executes, another version of the database is created. For example, immediately after compilation of the project TEST, there would be a set of project database files ending with the version number 00. Each time the TEST simulation is executed, new database files are created with incremental version numbers 01, 02, 03, etc. To view the results of the second execution of the TEST project, the user would enter "SEEDATA TEST 2."

Although the SeeData browser currently checks for and enforces the hardware requirements discussed above, it should be possible to relax those requirements. As noted in the discussion that follows, every feature of the program invoked with the mouse may be accessed through alternative keyboard commands. Also, although the current prototype requires a VGA card, most of the code is written in a virtual (or logical) coordinate system. Therefore, the prototype should be able to support alternative screen sizes and resolutions without redesigning the program.

SeeData uses pulldown and popup menus to guide the user through the selection and display of database elements. When the program is first invoked, a menu bar is displayed at the top of the screen and a status bar is displayed at the bottom of the screen. The menu bar has two choices: FILES and VIEWS. If a project was not specified when SeeData was invoked, the VIEWS selection will be disabled.

The status bar contains information about the currently selected project and the currently selected class or object.

There are four options on the pulldown menu for the FILES selection:

1. HELP—Provides information about the help features of the browser
2. DIRECTORY—Allows the current directory from which projects are selected to be changed to another directory in the host file system
3. PROJECT—Allows the selection of a new project from the projects found in the current directory
4. EXIT—Exits the program.

Each menu selection has an associated help message. When a menu item is selected, the user may bring up the help message by either pressing the right mouse button or by pressing the F1 function key on the keyboard. These messages provide a brief description of the functionality of each selection.

Under DOS, disk space is divided into a hierarchical structure of directories and files. At any given time there is a current working directory from which files may be accessed. The DIRECTORY selection allows the user to specify an absolute pathname to designate a new working directory. This option is important in the context of persistent databases because the viewer will display only project databases listed in the current working directory.

Within a given directory, project databases are stored as a set of four project files, each with an extension of the form f00, d00, i00, or n00,* where 00 is a number between 00 and 99. For example, the initial version of the FSTAM project would be represented as the following set of files: FSTAM.F00, FSTAM.D00, FSTAM.I00 and FSTAM.N00. When the PROJECT selection in the FILES menu is selected, a list of projects contained in the current working directory is displayed in a popup menu, and the user may select a new project to view. Projects are displayed as a project name followed by a version number in parentheses.

There are four options on the pulldown menu for the VIEWS selection:

1. MODULE—Displays the modular structure of the project in the form of a dependence graph
2. CLASS—Allows a class to be designated from those of the current project, then displays that class definition**
3. INSTANCE—Lists the instances of the current class, allows an instance to be specified, and displays the instance's field values
4. HIERARCHY—Displays the inheritance structure of the currently selected class.

* d = data, object attributes; i = index; f = schema definition of object types; n = additional information.

** A class definition corresponds to a ModSim OBJECT declaration.

The MODULE selection of the VIEWS menu constructs a graphical representation of the dependencies between modules in a given project. For example, the module hierarchy of the FSTAM project is depicted in Figure 10. Two things are worth noticing about this example: (1) a module's dependencies are only expanded once, and (2) each displayed module name can be selected with the mouse.

The dependency tree is built in a breadth-first fashion to ensure that the module dependencies are only expanded once. In other words, since DATASET (which is used by GLOBALVARS, which is used by FSTAM) is expanded on the third level of the hierarchy, the DATASET used by UNITDATASET on the seventh level is not expanded. Therefore, to view the dependencies of any particular module, the tree should be searched from the top down in a breadth-first manner to make sure that the first occurrence of the module is found. This ensures that all modules upon which this particular module depends will be displayed in the hierarchy.

The magnifying glass icon on the video display permits the user to inspect the object under the glass. By double-clicking the left mouse button on the module under the magnifying glass, a module information display is brought up. This display contains three pieces of information: (1) the type of module (MAIN, DEFINITION, or IMPLEMENTATION), (2) the modules upon which the current module depends, and (3) the classes (or objects) that this module defines.

The CLASS selection of the VIEWS menu displays information about a particular object. When this menu item is selected, a list of all the classes in the current database is displayed and the user is instructed to select a class to become the current class. When a class is selected, a class template is displayed. The class template is roughly equivalent to the original object declaration, which was used in the source code to define the class.

Once a class is selected as the current class, an instance of the class may be examined. If the database is not an initial database, there will be information recorded for each instance of an object that the source program defined. For instance, in the Ping program shown in Figure 11, two variables are declared to be of type Player: Man1 and Man2. These are instances of the class Player and may be examined in the browser. Figure 12 shows the display of the Man1 instance of class Player. From this display we can gather that Man1 hit the ball 110 times during the simulation.

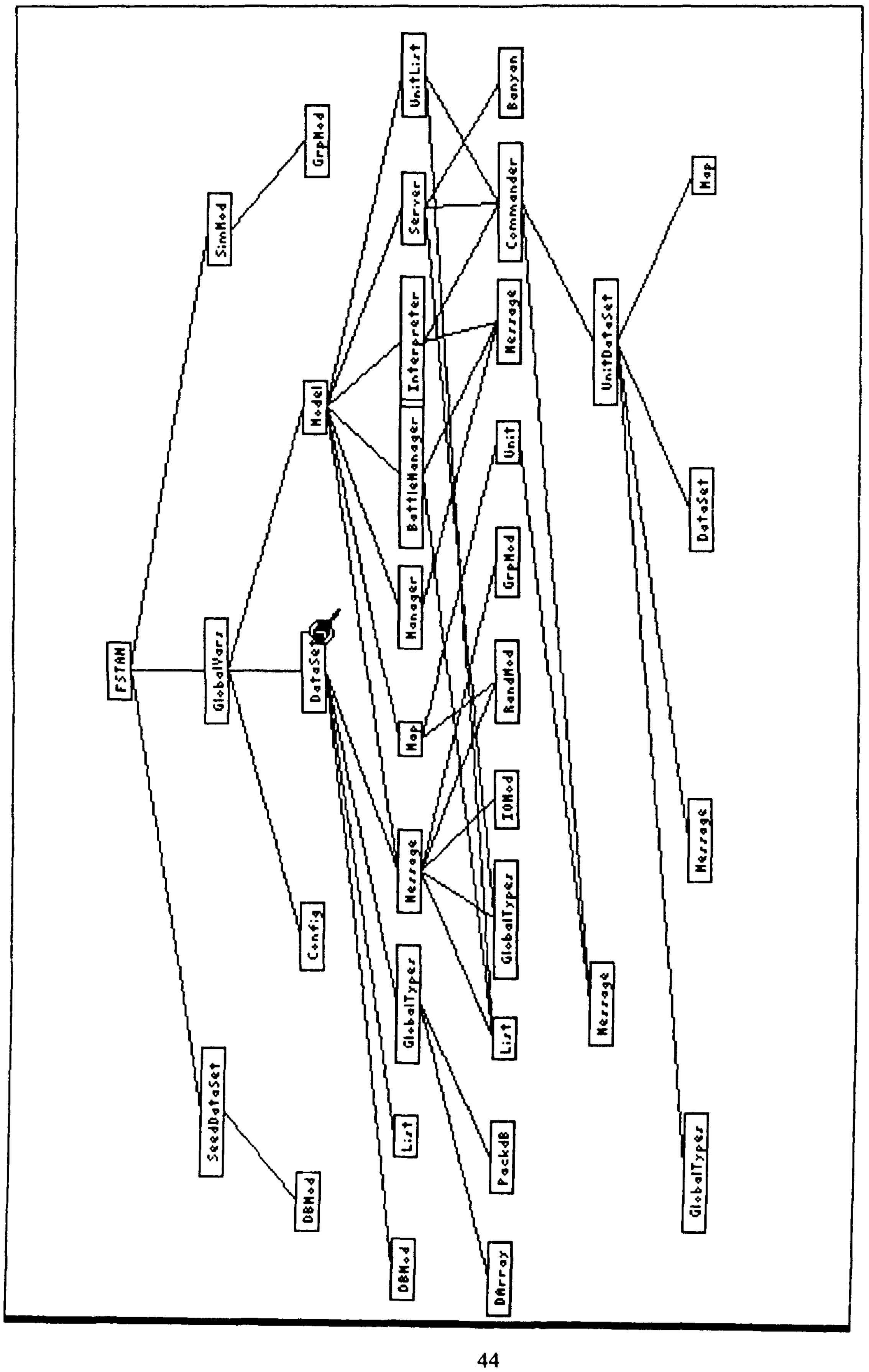
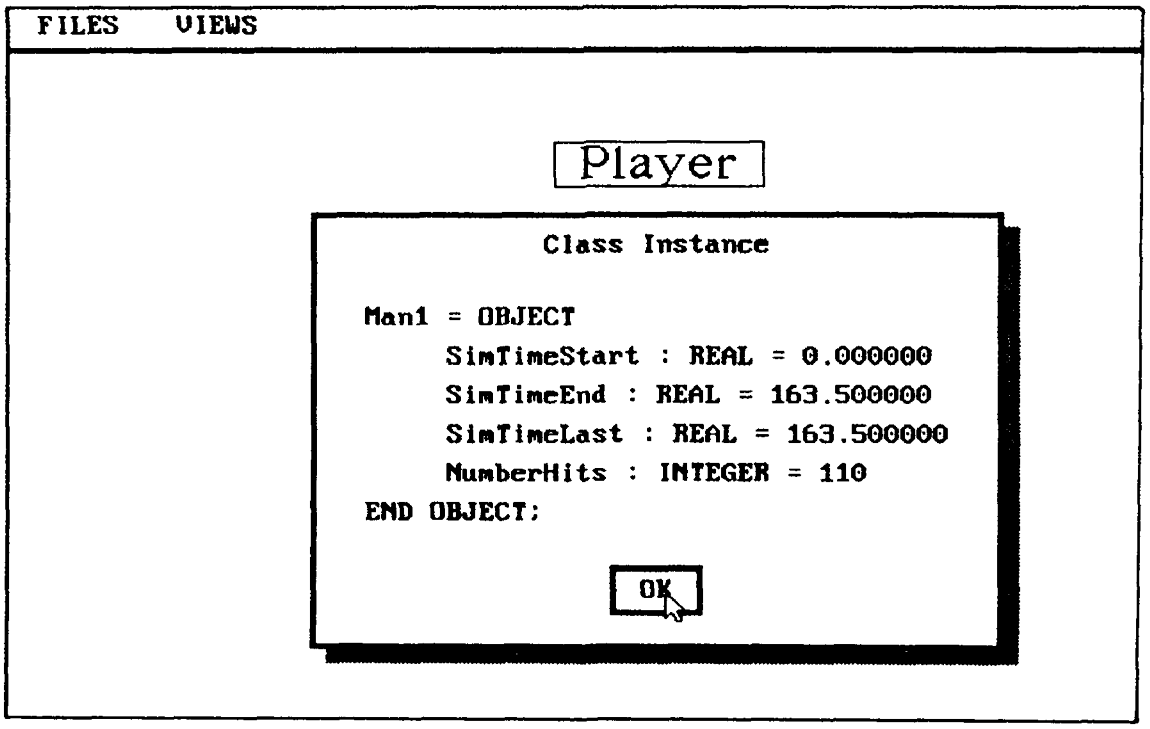


Figure 10. Module Hierarchy for the FSTAM combat model.

```
MAIN MODULE Ping;

FROM     SimMod IMPORT StartSimulation;

CONST    MaxHitsInGame = 110;

TYPE     Player = OBJECT
          NumberHits  :  INTEGER;
                  Opponent     :  Player;
                  ASK  METHOD MeetOpponent (IN Opponent : Player);
                  TELL METHOD PlayGame;
          TELL METHOD HitBall;
         END OBJECT;

VAR
    Man1, Man2 : Player;

BEGIN
    NEWOBJ (Man1);
    NEWOBJ (Man2);

    ASK Man1 TO MeetOpponent (Man2);
    ASK Man2 TO MeetOpponent (Man1);

    TELL Man1 TO PlayGame;
    TELL Man2 TO PlayGame;

    TELL Man1 TO HitBall;
    StartSimulation;

    DISPOSEOBJ (Man1);
    DISPOSEOBJ (Man2);

END MODULE.
```

Figure 11. ModSim code for the Ping process.



```
Man1 = OBJECT
      SimTimeStart : REAL = 0.000000
      SimTimeEnd : REAL = 163.500000
      SimTimeLast : REAL = 163.500000
      NumberHits : INTEGER = 110
END OBJECT;
```

Figure 12. Object instance Man1 from the Ping project.

The inheritance structure for a selected class may be displayed by choosing the HIERARCHY selection of the VIEWS menu. Like the modules hierarchy display, the class hierarchy objects are also mouse-sensitive. Double-clicking left on an object displayed in the class hierarchy causes that object class to become the currently selected class and displays the class template. Double-clicking right on an object in the class hierarchy causes that object class to become the currently selected class and displays class instances of the object.

Although useful in its own right, it should be noted that the current browser is only a primitive prototype that serves mainly as a demonstration of what is possible with a persistent language compiler.

Deficiencies of the Prototype

The persistent compiler and support tools described in this chapter represent a prototype version of ModSim with only rudimentary support for persistent objects. This section describes other features required to support full persistence in ModSim.

Complex Types

To be useful in production environments, a data management capability for persistent objects should be able to handle the storage and retrieval of complex types, such as records, pointers, and recursively defined types. The current prototype handles only object field variables declared to be basic types (e.g., INTEGER, REAL, BOOLEAN, etc.) and types derived from the basic types (e.g., subranges). It does not store complex types nor does it provide a mechanism for dereferencing pointer types (ModSim's ADDRESS and POINTER types). Solutions to this problem are greatly simplified by a consistent type system and a consistent approach to type representation within the compiler. Careful consideration must be given to the semantics of persistent objects in the presence of dynamic pointers and to the tradeoffs between a simplified type system versus the problems associated with attempting to store objects represented by dynamic pointers and complex types.

Modular Recompilation

The current prototype does not address the problem of updating class/instance definitions. Once they are formed in the database they are not revisited. If the object representations in the program are to be changed, the old database must be explicitly deleted and rebuilt through the compilation process to ensure consistency between the program's view and the database's view of the object structure. Future versions of the compiler must be able to manage change and incremental compilation of object definitions without requiring a complete recompilation of the program. This problem is complicated by the presence of inheritance in the object definitions. Because of dependencies among object definitions, each time a class is encountered during compilation it will need to be checked for consistency. If the database's representation for the class is different from the one under examination, a change has occurred, and that change must be propagated to all affected classes (i.e., those which inherit from the current class).

One strategy that works within the current framework is to generate instances from the class definitions at runtime rather than compile time. This would ensure a consistent object representation, but would cause a performance penalty. A more radical scheme would involve reworking the compiler so that the only representation of the object was within the database—in other words, to make the database the intermediate language for the compiler. The compiler would translate the source code into a network of database entities that would encode the representation of objects and their relationships. Object code could then be generated from the database through a variant of a database query.

Object-Instance Initialization

With the availability of persistent data, the possibilities for object instance initialization are expanded. When an object instance is created, normally the programmer will explicitly initialize each field to some constant value in the source code. With persistent data, a new object instance could be initialized to the same field values as an object instance in persistent storage (i.e., a new object ID with a copy of all the field values). Alternatively, an object instance could be instantiated during program execution and initialized to be the object instance from persistent storage (i.e., the same object ID with a copy of all the field values). The most extreme form of initialization might be to initialize each field from the same or different persistent object instances. The usefulness of such initialization approaches remains to be determined.

There are also several ways the initialization process could be achieved. A static method would be to retrieve the initial values from persistent storage during compilation and compile them into the program. A single object instance (or many object instances) could receive these initial values. A more dynamic method would be to compile a query into the program that actually retrieves the initial values from persistent storage at execution time when an object is created. This allows for flexible queries, such as retrieving the most recent object instance for an object type as the initial field values. Each execution of the program could retrieve different object instances with entirely different field values. The performance cost of the queries will need to be considered.

Query Facilities

Agrawal and Gehani[29] describe extensions to a persistent version of C++. Several features of their work are of interest to this effort, particularly their treatment of collections and querying. A query facility in ModSim will allow easy retrieval of persistent object instances into an executing program. An iteration operator in ModSim would appear as:

```
FOR O IN T
    [SUCHTHAT condition]
    [BY field]
      ...
END FOR
```

Where "O" is a variable of type object and "T" is the name of an object type. The statement is executed each time an object is found in the collection whose fields satisfy the SUCHTHAT expression. The expression of the BY field clause causes the loop iteration to examine stored objects in order of increasing values of the field specified in the expression. The iterator FORALL is added to permit accessing of class hierarchies. FORALL spans all classes descending from the class of the specified object. Otherwise it works the same as FOR. There is also the need for an IS operator to compare types of objects selected in the FORALL construct.

[29] R. Agrawal and N. Gehani, "Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language c++," *2nd International Workshop on Database Programming Languages* (Morgan Kaufmann, 1989).

The phototype naturally includes the class-hierarchy collection, but inclusion of subcollections is not proposed at this time. The inclusion of the FOR, FORALL and IS constructs provide the needed query operations at the language definition level. These are especially useful for initialization of stored objects from persistent storage. The addition of two fields to each object—revision number and simulation time-stamp—provides for complete control over access to stored objects.

Versioning

The researchers consider it extremely important to maintain a revision history for an object instance. This allows a single object instance to have various states stored and accessible. Various possibilities and issues in maintaining a revision history require further investigation. Versioning gives rise to a number of important unanswered questions:

1. *How are objects in a revision history referenced?* A revision history is an ordered list of states (versions) of the same object instance. A way to access a specific version is required that could be supported by storing a revision number, revision name, or revision timestamp with each version. If the list is a linear, ordered list, this would imply that a new version is only created from the most recent version.

2. *Is it valuable to allow a single version of an object to spawn multiple versions?* This would be a more complex arrangement where the revision history of an object instance takes the form of a tree.

3. *What modifications to an object instance should exist in the revision history?* One possibility is to save each change of state (i.e., field update) in the revision history. This could be done without any explicit direction from the programmer. For an object instance with many fields, a change that alters each field could result in many versions with only a small difference in each version. It may be more desirable in this case to have a new version created when the state change for the object is complete (i.e., all or many of its fields have been modified). To allow this flexibility, the creation of each version might have to be explicitly declared by the programmers.

4. *Are object instances from the same object type or different object types with a revision history related?* The revision history for a single object instance should clearly indicate the relationship the versions have with each other, most likely a time sequence. A group of related object types in a simulation might each have object instances with a revision history. It would be desirable to reference this group later as a logical entity.

5. *Can a referencing mechanism support this?* If each logical version for all objects has the same revision number, revision timestamp, etc., then the objects should be accessible as a group at a later time. Some useful cases for using a revision history need to be identified, and the design should support the widest range of possibilities feasible.

6 SUMMARY AND FUTURE DIRECTIONS

Summary

This interim report has presented the major thrusts of research into modeling and simulation, and the application of new technologies to the area of combat simulation. The work specifically addresses the representation of combat engineers in combat simulation. These efforts are summarized below.

Object-oriented programming is replacing structured programming as the dominant software paradigm. This movement will have a great impact on the design and production of simulations. It is fortunate that the object-oriented paradigm has its roots in simulation languages. The simulation community should play an active role in the development of object-oriented design techniques and languages suited to the production of quality simulations. The U.S. Army has taken such a role in creating the ModSim language.

The researchers have presented a design for Version 2.0 of the FSTAM combat simulation, which is based on the object-oriented approach using ModSim. An advantage of the object-oriented paradigm for combat modeling is the ability to localize important simulation components for ease of modification and maintainability. A goal of this design is to decouple the behavior of the simulation objects from the mechanics of the simulation. To achieve this goal, the design permits the attachment of autonomous intelligent decisionmaking agents to each combat unit in the simulation.

The development and operational use of large simulation models requires investigation into support environments. Hypertext is an attractive candidate technology for investigation. The researchers described the application of hypertext in creating an environment for FSTAM 1.0 and have provided general requirements for further research in this direction. Hypertext is still an emerging technology. Many challenging design and implementation issues must be addressed before hypertext systems can move from research environments to general acceptance. These include basic issues such as the underlying data model and the user interface. These issues must be addressed successfully if the benefits of hypertext-based modeling environments are to be realized.

The power of object-oriented design to facilitate the modeling of complex relationships among real-world entities makes it well suited to the creation of large simulations. Complex simulations, however, require large amounts of data for scenario construction and produce copious amounts of data as their result. Current simulation languages, including ModSim, do not provide secondary storage capabilities that are efficient and compatible with internal representations of objects. This report has presented a description of the development of a working prototype of a persistent ModSim compiler. This prototype compiler integrates ModSim with object-oriented data management facilities. This integration achieves a high degree of efficiency of data retrieval and storage, and supports a consistent approach to the manipulation and analysis of data associated with both scenario development and simulation results.

The data-management facility underlying the current prototype of persistent ModSim is very limited. Public-domain and commercially-available object-oriented database systems are currently being evaluated with the goal of selecting a data-management facility with sufficient power to permit the adding of needed capabilities to the current prototype of persistent ModSim. However, even with restricted functionality of the prototype, the investigation of persistent ModSim has provided insight into the potential benefits of integrating object-oriented data-management facilities and simulation technology. A ModSim compiler with a seamlessly integrated and consistent data-management facility could greatly contribute to the development and use of simulation, perhaps even fundamentally change the way simulations are created and used.

Future Directions

With the experience and insight gained from the application of object-oriented simulation using ModSim to the redesign of the FSTAM simulation, the investigation of hypertext for information management, and the development of a persistent ModSim prototype, the researchers can now make some strategic observations and chart a course for the future direction of this research.

Object-oriented databases and persistent programming languages are the enabling technologies that will permit the development of the next generation of complex data-driven applications such as computer-aided design, computer-aided software engineering, and geographic information systems. A major goal of the developers of these next-generation systems will be integration and interoperability. The researchers foresee a need for simulation in most of the systems. These simulations should run directly on the databases underlying these applications. This research is on the threshold of providing an environment wherein simulation (as well as optimization techniques and other higher-level tools) can be applied within next-generation integrated applications.

Chief among the benefits of an integrated data-management facility is an interface standard for the development of advanced support tools for scenario development and analysis. Instead of the laborious process of defining massive datasets for simulation of a specific situation, entities in the simulation could inherit their behaviors and their initial state from a library of persistent objects. Only those elements unique to the situation under analysis would be specified. Given tools that allow editing (such as an advanced version of the class-hierarchy browser), changing the parameters of the simulation becomes as easy as viewing a particular object instance and changing some or all of its instance variables.

Change management would track such changes and link causal relationships between simulation parameters and simulation outcome. Instead of megabytes of unstructured text and numbers being written into files, the results of the simulation are captured in the database and readily viewable. Specific questions and causal relations may be discovered by database queries and the use of intelligent analysis tools instead of the tedious process of manually scanning unnecessarily vast amounts of simulation output. Since all aspects of the simulation could conceivably be captured, the activity list and other internal structures that manage the simulation could be stored.

One future possibility is the building of tools to aid in the creation of simulations. These would be analogous to conventional debuggers for programming languages. A simulation debugger could be created by storing the causal relationships that trigger state changes in objects as recorded in the versioning information; this would allow the evolution of the state of objects in a simulation to be traced and reasoned about automatically. A byproduct of storing the internal simulation structures would be the ability to arbitrarily halt a simulation, later restarting it at the same place (possibly with changes to some state variables). Such an environment would represent a quantum leap forward in the state of combat simulation technology.

Areas of research that could support this effort include (1) the development methodolgies, analysis, and design of persistent object-oriented simulations using ModSim, and (2) extension of the current object-oriented paradigm within the context of persistent simulation.

REFERENCES

Agrawal, R., and N. Gehani, "Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language c++," *2nd International Workshop on Database Programming Languages* (Morgan Kaufmann, 1989).

Belanger, R., and S. Rice, *ModSim User's Manual* (CACI Products Company, 1988).

Bigelow, J., "Hypertext and CASE," *IEEE Software* (Institute of Electrical and Electronics Engineers [IEEE], March 1988), pp 23-27; A. Hecht, "The Hypertext Features of Teamwork Analysis and Design System," *Proceedings of the Second International Workshop on Computer-Aided Software Engineering*, Vol 1 (1986), pp 1-44.

Bratley, P., B. Fox, and L. Schrage, *A Guide to Simulation* (Springer-Verlag, 1983).

Buckle, J.K., *Software Configuration Management* (Macmillan Press, 1982).

Conklin, E.J., "Hypertext: An Introduction and Survey," *IEEE Computer*, Vol 2, No. 9 (1987), pp 17-41.

Dahl, O.J., B. Myrhaug, and K. Nygaard, *SIMULA 67 Common Base Language* (Norwegian Computing Center, Oslo, 1984).

DOD *Critical Technologies Plan* (DOD Office of the Deputy Under Secretary for Acquisition, 1990).

DOD Simulations: Improved Assessment Procedures Would Increase the Credibility of Results, GAO Program Evaluation and Methodology Division (PEMD) Publication 88-33 (GAO, December 1987), pp 47-53.

DOD-STD-2167, *Defense System Software Development* (DOD, June 1985), pp 2-3.

Dunnigan, J.F., and A. Bay, *A Quick and Dirty Guide to War* (William Morrow, 1982).

Elliot, J., and B. Moss, "Object-Orientation as Catalyst for Language-Database Integration," *Object-Oriented Concepts, Databases and Applications* (ACM Press, 1989).

Fairley, R.E., *Software Engineering Concepts* (McGraw-Hill Book Co., 1985).

Genesereth, M.R., and N.J. Nilson, *Logical Foundations of Artificial Intelligence* (Morgan-Kaufmann Publishers Inc., 1988).

Krasner. G.E., and S.T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *Journal of Object-Oriented Programming* (August/September 1988), pp 26-49.

Larson, N., *Hyperlink* (MaxThink, Inc., 1988).

Meyer, B., *Object-Oriented Software Construction* (Prentice Hall, 1988).

Mullarney, A., J. West, R. Belanger, and S. Rice, *ModSim Tutorial* (CACI Products Company, 1988).

Nance, R.E., *The Conical Methodology: A Framework for Simulation Model Development*, Technical Report SRC-87-002 (System Research Center, Virginia Tech, 1986).

Nelson, T.H., "Getting it Out of Our System," *Information Retrieval: A Critical Review* (Thompson Books, 1967), pp 191-120.

Parnas, D., "On the Criteria to be Used in Decomposing Modules," *Communications of the ACM*, Vol 15, No. 2 (1972), 1053-58.

Snellen, J.E., *The MALOS Combat Engineer Simulation Environment (Version 3.0)*, Vol 1 and Vol 2, Technical Report (TR) P-86/02/ADB105887/ADB105862 (U.S. Army Construction Engineering Research Laboratory [USACERL], August 1986).

Subick, C., and M.A. Fiddes, *Force Structure Tradeoff Analysis Model (FSTAM) User's Manual*, Automatic Data Processing (ADP) Report P-88/24/ADB127731L (USACERL, August 1988).

Taylor, J.G., *Lanchester Models of Warfare* (Military Applications Section, Operations Research Society of America, 1983).

Ullman, J., *Principles of Database and Knowledge-Base* (Computer Science Press, 1988).

Widman, L.E., and K.A. Loparo, "Artificial Intelligence, Simulation and Modeling: A Critical Survey," *Artificial Intelligence, Simulation and Modeling*, L.E. Widman, K.A. Loparo, N.R. Nielson, eds. (John Wiley and Sons, 1989).

Wirth, N., *Programming in Modula-2* (Springer-Verlag, 1982).

Zdonik, S., and D. Maier, *Readings in Object-Oriented Databases* (Morgan Kaufmann, 1989).

ABBREVIATIONS

ACM	Association of Computing Machinery
AI	artificial intelligence
APP	automatic data processing
CASE	computer-aided software engineering
DBMS	database management system
DOD	Department of Defense
FSTAM	Force Structure Tradeoff Analysis Model
GAO	General Accounting Office
MISMA	U.S. Army Model Improvement and Studies Management Agency
ModSim	Modular Simulation Language
PEMD	Program Evaluation and Methodology Division
SQL	structured query language
TR	Technical Report
USAES	U.S. Army Engineer School
VGA	virtual graphics adapter

DISTRIBUTION

Chief of Engineers
ATTN CEHEC-IM-LH (2)
ATTN CEHEC-IM-LP (2)
ATTN CERD-L
ATTN DAEN-ZCM

US Army Engineer School
(Ft. Leonard Wood, MO)
ATTN: ATSE-CDM-T (5)
ATTN: ATSE-TD (5)

US Army Combined Arms Center
ATTN: ATZL-CAC (5)

Topographic Engineering Center
ATTN: CETEC-GL-A

Director, CECEM-MCSD
ATTN: AMSEL-RD-SE-MCS

US Army Corps of Engineers
ATTN: CEHND-ED-SY

Naval Civil Engr Lab
ATTN: Library 93402

Army War College
ATTN: Library 17013

US Army Command & General Staff College
ATTN: ATZL-SWH 66027

CEWES, ATTN: Library 39180
ATTN: CEWES-EN
ATTN: CEWES-GA

Engineer Studies Center
ATTN: Library 22060

Defense Technical Info Center
ATTN: DTIC-FAB (2)